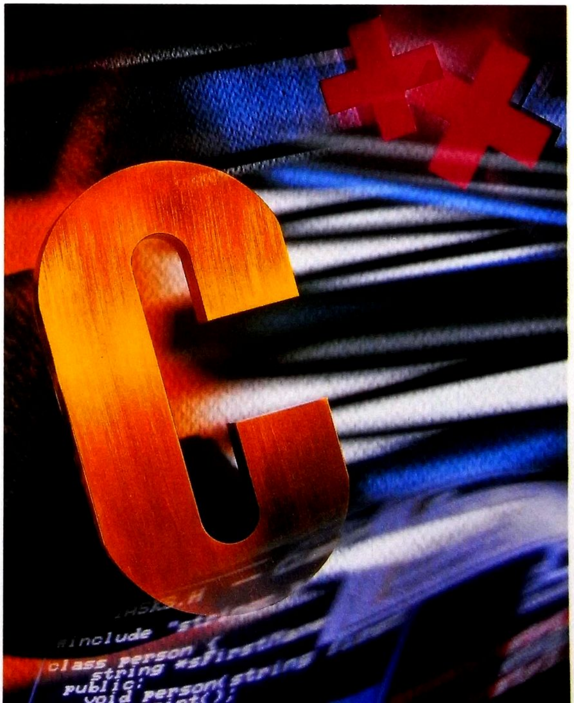
OBJECTWINDOWS™ FOR C++

PROGRAMMER'S GUIDE



BORLAND

ObjectWindows™ for C++

プログラマーズガイド

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001

編訳　ボーランド株式会社 1991

目　次

はじめに …… 1

ObjectWindows のパッケージ …… 2

ObjectWindows マニュアルの内容 …… 2

ハードウェア，ソフトウェア構成 …… 3

ObjectWindows のインストール …… 4

INSTALL の使用 …… 4

ObjectWindows の README ファイル …… 4

ObjectWindows のヘルプファイル …… 5

Windows のヘルプ …… 5

DOS のヘルプ …… 6

マニュアルの表記 …… 7

ObjectWindows の命名規約 …… 8

製品に関するお問い合わせ …… 9

参考書籍 …… 9

第1部　ObjectWindows とは …… 11

第1章　ウィンドウの継承 …… 13

Windows アプリケーションとは …… 13

Windows の有益性 …… 15

ハードウェアの条件 …… 16

オブジェクト指向のウィンドウ処理 …… 16

Windows に適したインターフェース …… 16

ウィンドウ情報のカプセル化 …… 17

Windows 関数の抽象化 …… 18

メッセージ応答の自動化 …… 18

Windows プログラムの構造 …… 23

Windows の構造 …… 24

Windows および DOS との対話 …… 24

“Hello, World” …… 25

アプリケーション起動時の処理 ……26
メインウィンドウの役割 ……27
アプリケーション開発サイクル ……28

第2章　Windowsを使ってみよう ……29
ObjectWindowsアプリケーションの構築：予備知識 ……30
コンテナクラスライブラリ ……30
ディレクトリ ……31
正しいライブラリの指定 ……32
DLLを使うObjectWindowsアプリケーション ……32
リソースファイルの作成 ……33
ObjectWindowsアプリケーションの構築：詳細 ……34
IDEでObjectWindowsアプリケーションを構築する場合 ……34
コマンドラインツールでObjectWindowsアプリケーションを構築する場合 ……36
ステップ1：簡単なWindowsアプリケーション ……37
アプリケーションの必須条件 ……37
アプリケーションクラスの定義 ……38
ステップ2：メインウィンドウクラス ……41
ウィンドウオブジェクトとは ……41
ハンドル ……42
親子関係 ……42
メインウィンドウオブジェクトの作成 ……43
メッセージへの応答 ……44
アプリケーションの終了 ……46

第3章　ウィンドウへの書き込み ……49
ディスプレイコンテキストとは ……49
ステップ3：ウィンドウにテキストを描画する ……51
メッセージ構造体 ……52
画面のクリア ……53
ステップ4：ウィンドウに線を描画する ……53
ドラッグモデル ……54
ドラッグメッセージへの応答 ……55
ステップ5：線の太さを変更する ……57
新しいペンの選択 ……58

ペンの変更 ……60
入力ダイアログの実行 ……60
ステップ6：ウィンドウを再描画する ……62
ペイントモデル ……62
オブジェクトとしてのグラフィックスの格納 ……63
格納してあるグラフィックスの再描画 ……66

第4章　メニューの追加 ……69
メニューリソース ……69
ステップ7：メニューを追加する ……71
メニューメッセージの受け取り ……72
メニューメッセージへの応答 ……73

第5章　ダイアログの追加 ……75
ステップ8：ポップアップウィンドウとダイアログボックスを追加する ……76
ポップアップウィンドウの作成 ……77
MakeWindow 関数 ……78
ダイアログボックスの追加 ……78
データメンバの追加 ……80
ダイアログボックスの実行 ……81
ステップ9：描画内容をファイルに格納する ……82
状態の監視 ……82
ファイルのオープンとセーブ ……85

第6章　ウィンドウのポップアップ ……87
ステップ10：ヘルプウィンドウのポップアップ ……87
ObjectWindows でモジュールを使う ……88
メインプログラムの変更 ……88
モジュールの作成 ……89
ウィンドウにコントロールを追加する ……91
コントロールの実体 ……91
ウィンドウコントロールの作成 ……92
データメンバとしてのコントロールオブジェクト ……93
コントロールの管理 ……94
コントロールイベントへの応答 ……96

第2部　ObjectWindows を使う ……99

第7章　ObjectWindows の概要 ……101

ObjectWindows の規約 ……101
ObjectWindows の階層 ……102
Object ……104
TModule ……104
TApplication ……104
インターフェースオブジェクト ……105
TWindowsObject ……105
ウィンドウオブジェクト ……105
TWindow ……105
TEditWindow ……106
TFileWindow ……106
ダイアログオブジェクト ……106
TDialog ……106
TFileDialog ……106
TInputDialog ……107
コントロールオブジェクト ……107
TControl ……107
TButton ……107
TCheckBox ……107
TRadioButton ……107
TListBox ……108
TComboBox ……108
TGroupBox ……108
TStatic ……108
TEdit ……108
TScrollBar ……108
MDI オブジェクト ……109
TMDIFrame ……109
TMDIClient ……109
スクローラオブジェクト ……109
Windows API 関数 ……110
ObjectWindows による Windows 関数の呼び出し ……110

Windows 関数へのアクセス ……110
スタイル定数の組み合せ ……112
Windows 関数の種類 ……112
ウィンドウマネージャインターフェース関数 ……112
グラフィックスデバイスインターフェース（GDI）関数 ……113
システムサービスインターフェース関数 ……113
コールバック関数 ……113
列挙関数 ……114
スマートコールバックの使用 ……115
Windows メッセージ ……116
Windows メッセージの引数 ……117
Windows メッセージの種類 ……118
ウィンドウ管理メッセージ ……118
初期化メッセージ ……118
入力メッセージ ……118
システムメッセージ ……119
クリップボードメッセージ ……119
システム情報メッセージ ……119
コントロール操作メッセージ ……119
コントロール通知メッセージ ……119
スクロールバー通知メッセージ ……120
クライアント領域外メッセージ ……120
マルチドキュメントインターフェース（MDI）メッセージ ……120
デフォルトのメッセージ処理 ……120
メッセージの送出 ……121
メッセージ範囲 ……122
ユーザ定義メッセージ ……122

第 8 章　モジュールとアプリケーションオブジェクト ……125
アプリケーションの流れ ……125
アプリケーションの初期化 ……126
メインウィンドウの初期化 ……127
各アプリケーションインスタンスの初期化 ……129
最初のアプリケーションインスタンスの初期化 ……129
アプリケーションの実行 ……132

アプリケーションのクローズ ……132

第9章　インターフェースオブジェクト ……135
TWindowsObject ……135
インターフェースオブジェクトが必要な理由 ……136
ウィンドウの親子関係 ……137
子ウィンドウリスト ……138
子ウィンドウの繰り返し処理用メンバ関数 ……139
メッセージ処理 ……140
メッセージへの応答 ……141
コマンドメッセージと子ウィンドウメッセージ ……142
コマンドメッセージ処理 ……143
子メッセージの処理 ……144
デフォルトのメッセージ処理 ……144

第10章　ウィンドウオブジェクト ……147
TWindow クラス ……148
ウィンドウオブジェクトの初期化と作成 ……148
ウィンドウオブジェクトの初期化 ……148
ウィンドウ要素の作成 ……152
初期化と作成のまとめ ……152
ウィンドウクラスの登録 ……153
登録属性 ……155
クラススタイルメンバ ……156
アイコンメンバ ……156
カーソルメンバ ……156
バックグラウンド色メンバ ……157
デフォルトメニューメンバ ……157
ウィンドウのスクロール ……157
スクローラの属性 ……158
ウィンドウにスクローラを追加する ……159
スクロールの例 ……160
自動スクロールとトラッキング ……161
スクロール単位と範囲の変更 ……162
スクロール位置の変更 ……163

ページサイズの設定 ……164
メンバ関数 Paint の最適化 ……164
Edit ウィンドウと File ウィンドウ ……165
Edit ウィンドウ ……166
File ウィンドウ ……168

第11章　ダイアログオブジェクト ……171
ダイアログリソースの使用 ……171
子ダイアログオブジェクトの使用 ……172
子ダイアログオブジェクトの構築と初期化 ……172
ダイアログの作成と実行 ……173
モードつきダイアログとモードなしダイアログ ……174
子ダイアログのクローズ ……174
ダイアログをメインウィンドウとして使う ……175
モードなしダイアログ用の Windows クラスの定義 ……176
コントロールの操作とメッセージ処理 ……176
ダイアログコントロールの操作 ……177
コントロール通知メッセージに対する応答 ……179
ダイアログとコントロールの通信例 ……180
ダイアログ機能の拡張例 ……181
入力ダイアログ ……181
ファイルダイアログ ……183

第12章　コントロールオブジェクト ……185
コントロールオブジェクトの使用 ……186
コントロールの構築と作成 ……187
コントロールの破壊と削除 ……188
コントロールとメッセージ処理 ……188
ウィンドウ内のコントロールの操作 ……188
コントロール通知メッセージに対する応答 ……189
コントロールフォーカスとキーボード ……191
リストボックスコントロール ……191
リストボックスの構築と作成 ……191
リストボックスの変更 ……192
リストボックスの照会 ……194

リストボックスからの選択項目の取得 ……195
コンボボックス ……197
3種類のコンボボックス ……197
シンプルコンボボックス ……198
ドロップダウンコンボボックス ……198
ドロップダウンリストコンボボックス ……198
コンボボックスの選択 ……199
コンボボックスの構築 ……199
コンボボックスの変更 ……200
アプリケーション例：CBoxTest ……200
スタティックコントロール ……201
スタティックコントロールの構築 ……202
スタティックコントロールの照会 ……203
スタティックコントロールの変更 ……203
アプリケーション例：StatTest ……203
エディットコントロール ……203
エディットコントロールの構築 ……204
クリップボードと編集操作 ……206
エディットコントロールの照会 ……207
エディットコントロールの変更 ……210
テキストの削除 ……210
テキストの挿入 ……210
テキスト選択とスクロールの強制 ……211
プログラム例：EditTest ……211
プッシュボタンコントロール ……212
ボタンメッセージへの応答 ……212
チェックボックスとラジオボタン ……213
チェックボックスとラジオボタンの構築 ……213
選択ボックス状態の照会 ……214
選択ボックス状態の変更 ……214
チェックボックスメッセージとラジオボタンメッセージへの応答 ……215
グループボックス ……216
グループボックスの構築 ……216
グループボックスメッセージへの応答 ……216
プログラム例：BtnTest ……217

スクロールバー ……218
スクロールバーオブジェクトの構築 ……219
スクロールバーの照会 ……220
スクロールバーの変更 ……220
スクロールバーイベントへの応答 ……221
プログラム例：SBarTest ……223
コントロールデータの転送 ……224
転送バッファの定義 ……224
コントロールの構築と転送機構の使用 ……227
データの転送 ……228
コントロール転送のカスタマイズ ……229
転送例 ……229

第 13 章　カスタムコントロールオブジェクト ……231
定義済みコントロールの変更 ……231
作成スタイルの変更 ……231
定義済みのコントロールを描画可能にする ……232
定義済みのメッセージ応答の変更 ……233
定義済みコントロールに対する追加処理 ……233
定義済みコントロールの応答のオーバーライド ……234
カスタムコントロールの使用 ……235
カスタムコントロールの設計 ……235
カスタムコントロールの定義 ……235

第 14 章　MDI オブジェクト ……239
MDI アプリケーションの要素 ……239
オブジェクトとしての MDI ウィンドウ ……241
MDI ウィンドウの構築 ……241
MDI フレームウィンドウの構築 ……242
MDI 子ウィンドウの構築 ……242
MDI アプリケーションでのメッセージ処理 ……244
MDI 子ウィンドウの管理 ……244
子ウィンドウの起動 ……244
子ウィンドウメニュー ……245
MDI アプリケーションの例 ……245

第15章　ストリームオブジェクト ……247
iostream ライブラリ ……248
オーバーロード演算子<<と>> ……249
ストリームクラスと TStreamable ……251
ストリームマネージャ ……252
ストリームクラスのコンストラクタ ……254
ストリームクラス名 ……255
ストリームマネージャの使用 ……256
　ストリームマネージャコードでリンクする ……256
　ストリームオブジェクトの作成 ……257
　ストリームオブジェクトの使用 ……258
ストリームにおけるコレクション ……258
　配列をストリームクラスにする ……258
　ストリームクラスのビルダ関数 ……259
　streamableName メンバ関数 ……261
　ストリームクラスのリーダ関数 ……262

第3部　Windows プログラミング ……265

第16章　メモリ管理 ……267
メモリマネージャの使い方 ……267
Windows のメモリ管理 ……268
　ハンドルとアドレス ……269
　　リアルモードの留意点 ……270
　　プロテクトモードの留意点 ……270
ローカルメモリとグローバルメモリ ……270
ローカルヒープの使い方 ……271
　ローカルメモリの確保と利用 ……272
　ローカルメモリブロックの解放と廃棄 ……273
　ローカルメモリブロックの再確保と変更 ……274
　ローカルメモリブロック情報の取得 ……275
　プログラミング上の留意点 ……276
グローバルヒープの使い方 ……276
　グローバルメモリの確保と利用 ……276
　グローバルメモリブロックをロックする別の方法 ……278

グローバルメモリブロックの解放と廃棄 ……279
グローバルメモリブロックの再確保と変更 ……279
グローバルメモリブロック情報の取得 ……280
グローバルメモリブロックの廃棄方法の変更 ……281
メモリ不足に関する警告の受理 ……281
プログラミング上の留意点 ……281

第17章　プロセス間通信 ……283
Windowsのクリップボード ……283
クリップボードのデータ形式 ……283
クリップボードへのデータの格納 ……284
クリップボードからのデータの取り込み ……286
レンダリングの遅延 ……288
プロセス間でのメッセージ伝達 ……289
ダイナミックデータエクスチェンジ ……290
条件 ……290
対話の成立 ……291
対話の終了 ……294
データ交換に使うメンバ関数 ……295
単一データ項目の要求 ……295
継続データ転送の維持 ……299
サーバへのデータ値変更要求 ……300
サーバでのマクロコマンドの実行 ……301
システムトピック ……303

第18章　GDIについて ……305
ディスプレイコンテキスト ……306
ディスプレイコンテキストの管理 ……306
ディスプレイコンテキストの構成要素 ……307
ビットマップ ……307
色 ……307
マッピングモード ……308
クリッピング領域 ……308
描画ツール ……309
描画ツール ……309

ストックツール ……310
論理ツール ……310
論理ペン ……311
論理ブラシ ……312
論理フォント ……313
ウィンドウでのグラフィックス表示 ……317
ウィンドウでの描画 ……318
ディスプレイコンテキストの操作 ……318
Windows グラフィック関数の呼び出し ……319
ウィンドウのペイント ……319
グラフィックスの描画方法 ……320
描画ツールの使い方 ……321
GDI 描画関数 ……323
文字列の描画 ……323
線の描画 ……324
MoveTo と LineTo ……324
連続線 ……325
弧 ……326
図形の描画 ……326
矩形 ……327
面取り矩形 ……327
楕円 ……328
扇型と弦 ……328
多角形 ……329
パレットの使い方 ……330
パレットの設定 ……331
パレットを用いた描画 ……332
パレット情報の取得 ……332
パレットの変更 ……333
パレット変更に対する応答 ……334

第 19 章　リソースの詳細 ……335
リソースの作成 ……336
実行形式ファイルへのリソースの追加 ……336
リソースコンパイラの実行 ……337

リソースコンパイラスクリプトファイル …… 338
リソースコンパイラ使用上の注意 …… 341
アプリケーションへのリソースの読み込み …… 342
メニューリソースの読み込み …… 343
アクセラレータテーブルの読み込み …… 344
ダイアログボックスの読み込み …… 345
カーソルとアイコンの読み込み …… 345
文字列リソースの読み込み …… 347
ビットマップの読み込み …… 348
ビットマップによるブラシの作成 …… 350
ビットマップのメニュー内表示 …… 351

第20章　アプリケーション作成の指針 …… 353

設計の原則 …… 353
ユーザの期待どおりの反応の提供 …… 354
ユーザによる作業工程の制御 …… 354
標準外観と標準動作 …… 355
一般的な事柄 …… 355
マウスとキーボードの対応関係 …… 356
メニュー …… 357
ダイアログボックス …… 359
設計時のその他の注意点 …… 359
安全なプログラムの書き方 …… 360
アトム操作 …… 360
セーフティプール …… 361
ウィンドウ作成に関する他のエラー …… 363
大需要オブジェクト …… 364

索引 …… 365

はじめに

ObjectWindowsはMicrosoft Windows用のアプリケーションの開発方法を一変させるものです。ほとんどのWindowsのプログラミングでは，ごく最近までMicrosoftのCコンパイラと何種類もの複雑な開発ツールを使っていたため，開発に膨大な時間がかかるばかりでなく，複雑で混乱を招くケースも発生していました。ところがObjectWindowsが提供するアプリケーション構築用フレームワークを利用するとWindowsのプログラミングがはるかに簡単になります。

Windowsには，いままで考えたこともなかったような新しいアイデアが盛り込まれています。たとえば，サイズを変更できるウィンドウでのテキストやグラフィックスの操作，マルチタスク環境下での他のプログラムとの対話処理，Windows API（アプリケーションプログラムインターフェース）での約600個の関数の操作などです。プログラムをWindowsアプリケーションとして開発する際に一番厄介なことは，何がWindowsアプリケーションとして必須事項であるかを認識し，必須事項をすべて盛り込まなくてはならないことです。

ObjectWindowsはこのような面倒な作業を一手に引き受けてくれます。ObjectWindowsは，Windowsアプリケーションに共通する，アプリケーションレベルとウィンドウレベルの動作をカプセル化した，オブジェクト指向のクラスライブラリなのです。次の特徴を備えています。

- Windowsとの一貫した，明快で，シンプルなインターフェース
- ウィンドウ管理とメッセージ処理用の標準装備の動作
- Windowsアプリケーションを構築するための基本フレームワーク

以上を共通の基礎機能として継承すれば，独自のアプリケーションに関する必要事項だけに専念することができるので，プログラマは一段と作業がやりやすくなります。

ObjectWindowsが提供するクラスは，Windowsアプリケーションの論理的な単位をひとつのこらずカプセル化するものではありません。むしろ，

ObjectWindowsが提供するクラスを再利用することによってアプリケーションの開発時間を大幅に短縮し，プログラム自体も簡略化しようという主旨のものです。

ObjectWindowsのパッケージ

ObjectWindowsは，1組のマスターディスクと次の2冊のマニュアルでパッケージになっています。

- ObjectWindows for C＋＋プログラマーズガイド（本書）
- ObjectWindows for C＋＋リファレンスガイド

利用者に便利なように，あらかじめ構築したObjectWindowsライブラリもディスクに入れてあります。

ディスクに入っているObjectWindowsのソースコードを使えば，Windowsアプリケーションをオブジェクト指向のObjectWindowsで記述できます。**ObjectWindows for C＋＋プログラマーズガイド**に載っているすべてのサンプルプログラムと自習用のサンプルプログラムもディスクに入っています。次の2つのヘルプファイルも入っています。

- WindowsのヘルプアプリケーションIM用のOWLWHELP.HLP
- DOS THELP用のOWLTHELP.OWH

ObjectWindowsマニュアルの内容

この**ObjectWindows for C＋＋プログラマーズガイド**では，ObjectWindowsで用いる新手法を詳しく解説しています。マニュアルは次の3部構成になっています。

- 第1部『**ObjectWindowsとは**』
  Windowsアプリケーションを記述するときの原則を説明します。基本的な説明の後に，ObjectWindowsアプリケーションの記述，拡張工程を段階的に修得できるチュートリアルが続きます。

- 第 2 部『**ObjectWindows を使う**』

  ObjectWindows そのものの各要素を詳しく説明します。ここでは，クラスの階層構造について概要を述べ，さらに Windows 環境との相互関係を解説します。

- 第 3 部『**Windows プログラミング**』

  Windows プログラミングの各局面のうち，グラフィックスなどの ObjectWindows によってカプセル化されない部分を概説します。プロセス間通信，リソースの利用，メモリ管理，コードとデータの共有についても説明します。Windows プログラミングのガイドラインもあります。

ObjectWindows for C++リファレンスガイドは，ObjectWindows のクラス，型，定数と Windows の関数，型，定数に関する参考資料です。参照しやすいようにアルファベット順に配列してあります。

ハードウェア，ソフトウェア構成

ObjectWindows は Windows アプリケーションを記述するための，アプリケーションフレームワークです。したがって，ObjectWindows アプリケーションの基本的な機器構成は Windows アプリケーションと同じです。最小機器構成として以下の装置が必要です。

- ハードディスク
- 2 MB 以上のメモリ
- Windows 対応のグラフィックスディスプレイ
- スタンダードモードまたは 368 エンハンスモードの Windows 3.0 以降

Windows アプリケーションを扱える Borland の C++コンパイラであればどれでも，ObjectWindows アプリケーションをそのソースコードも含めてコンパイルできます。

ObjectWindows のインストール

ObjectWindows には，INSTALL という自動インストール用のプログラムがついています。ファイルは圧縮されているので，ハードディスクにそのままコピーすることはできません。必ず，INSTALL プログラムを使ってください。このプログラムは ObjectWindows ファイルのコピーと復元を自動的に行ないます。なお，インストールディスクにある FILELIST.DOC ファイルで，全ファイルの一覧を見ることができます。

マスターディスクはすべてコピーしてから作業を始めてください。マスターディスクは万一に備えて別に保管しておきます。

ObjectWindows のパッケージに入っている同意書をよく読み，製品登録カードに記入して郵送してください。ObjectWindows の更新情報をお知らせします。

INSTALL の使用

INSTALL は，ObjectWindows ファイルの格納用ディレクトリをハードディスクに設け，そこにフロッピーディスク内のすべてのファイルをすべて転送します。次の方法でインストールしてください。

1. フロッピードライブにインストールディスク（ディスク 1)を挿入し，次のコマンドをタイプしてから**リターン**を押します（d は，ディスクを入れたドライブ名です)。

   d:INSTALL

2. インストールの画面が表示されたら，**リターン**を押します。
3. この後はプロンプトの指示に従ってください。

ObjectWindows の README ファイル

README ファイルには，マニュアルに盛り込めなかった最新情報が入っています。

README ファイルは次の方法で読みます。

1. フロッピードライブに ObejctWindows のインストールディスクを挿入します。ObjectWindows をすでにインストールしている場合は，ステップ3に飛んでください。
2. d：とタイプして**リターン**を押します。
3. README とタイプして**リターン**を押します。ファイルが表示されたら，↑と↓の矢印キーでファイルをスクロールして内容を見ることができます。
4. ***ESC*** で終了します。

ObjectWindows のヘルプファイル

ObjectWindows には，OWLWHELP.HLP と OWLTHELP.OWH の2つのヘルプファイルがあります。OWLWHELP.HLP は Windows 環境で，OWLTHELP.OWH は DOS 環境でヘルプを提供します。この2つのヘルプの内容は基本的に同じで，形式だけが異なります。

Windows のヘルプ

OWLWHELP.HLP は Windows のヘルプアプリケーション WINHELP.EXE 用のヘルプファイルです。プログラムマネージャアイコンを次の特性で作成しておくと便利です。

- **記述**：`ObjectWindows Help`
- **コマンドライン**：`WINHELP.EXE A:¥BORLANDC¥OWL¥OWLWHELP.HLP`

このアイコンをダブルクリックすれば，WINHELP.EXE の実行中に OWLWHELP.HLP が自動的にロードされます。ただし，ObjectWindows をデフォルトのディレクトリにインストールしていないと，OWLWHELP.HLP のパス名をそのまま使えない場合があります。

DOS のヘルプ

THELP はメモリ常駐型のユーティリティであり，DOS アプリケーションの実行中に ObjectWindows のオンラインヘルプを提供します。たとえば，Borland C++の IDE 環境で ObjectWindows アプリケーションを開発しているとき，プログラミングを中断せずに THELP を読み込んで ObjectWindows のオンラインヘルプを使えます。

THELP を読み込むには，DOS のコマンドラインに次のように入力します。

```
THELP /FA:¥BORLANDC¥OWL¥OWLTHELP.OWH
```

デフォルトのホットキーを変更できます。Borland C++ユーザーズガイド第 7 章を参照してください。

このコマンドによって THELP が読み込まれ，OWLTHELP.OWH ファイルが Turbo ヘルプファイルになります。読み込ませた THELP を呼び出すには，***HELP*** キーを押します。すると THELP が表示され，カーソル位置か，その近くにある単語がヘルプ索引上で検索されます。単語が見つかると，そのトピックのヘルプ画面が表示されます。単語が見つからない場合はヘルプインデックスがかわりに表示されるので，自分でトピックを選択することができます。

THELP の機能を以下にまとめます。

キー	動作
→←↑↓	カーソルを移動する。
ROLLUP / ROLLDOWN	カーソルを移動する。
SHIFT →←↑↓	カーソルを移動しブロックをマークする。
TAB	カーソルを次のキーワードに移動する。
SHIFT-TAB	カーソルを直前のキーワードに移動する。
GRPH ←	行頭に移動する。
GRPH →	行末に移動する。
リターン	現在のヘルプ画面でハイライト表示されているキーワードのヘルプ項目を選択する。
ESC	THELP を終了する。
SHIFT-F 1	ヘルプインデックスを表示する。
GRPH-F 1	見終わったヘルプ画面の最後のものから 20 画面を逆順に表示する。
CTRL-P	アプリケーションにサンプルテキストを貼り付ける。

コマンドラインの引数の指定を変えるだけで，多くのTHELPオプションを変更できます。詳細については**Borland C++ユーザーズガイド**第7章を参照してください。

マニュアルの表記

`Monospace type` この書体が示すのは，画面上またはプログラム内に現われるテキストです。タイプする内容を示すのにも使います。

[] このカッコは，テキスト内またはDOSコマンドライン内で使われます。カッコの内容は，環境によって異なる選択項目です。したがって，**このカッコとカッコ内のテキストを文字どおりにタイプしてはいけません**。

Bold Italics ObjectWindowsの関数，定数，変数を表わします。

Boldface この書体を使うのは，関数名，クラス，構造体名です。コマンドラインツールオプション(-W など）にも使います。

Gothic CとC++の予約語を表わします。

Italics イタリック体が示すのは，テキストに現われるデータ識別子です。データ識別子として変数名とクラスのデータメンバ名があります。そのまま使うか，または独自の名称に改めて使う単語を表わします。

Keycaps この書体は，キーボードのキーを表わします。「***ESC*** を押してメニューを終了します。」のように使います。

☞ 読者に特に注目してもらいたい箇所を指します。

ObjectWindows の命名規約

ObjectWindows の階層に属するクラスと構造体の名前は，***TWindow*** のように *T* で始まります。さらに，*T* で始まるクラス名のすべてに，次の関連型が定義されています。

- *Pclass* はクラス *class* を指すポインタ型です。たとえば，***PTWindow*** は ***TWindow*** を指すポインタとして定義されています。
- *Rclass* はクラス *class* への参照型です。たとえば，***RTWindow*** は ***TWindow*** への参照です。
- ***RPclass*** はクラス *class* を指すポインタへの参照です。たとえば，***RPTWindow*** は ***TWindow*** を指すポインタへの参照です。
- ***PCclass*** は const クラス *class* を指すポインタ型です。たとえば，***PCTWindow*** は const ***TWindow*** を指すポインタです。
- ***RCclass*** は const クラス *class* への参照型です。たとえば，***RCTWindow*** は const ***TWindow*** への参照です。

以上の型によって，Windows の複雑なプログラミング工程の一部が簡単になっています。以上の型を使わないと，TWindow へのポインタが必要な場合，Windows の DLL（ダイナミックリンクライブラリ）で次のような宣言が必要になります。

PTWindow に相当します。

```
Twindow _FAR *
```

あるいは，以下のように ***TWindow*** を指すポインタへの参照をつけます。

RPTWindow に相当します。

```
Twindow _FAR * FAR &
```

P，***R***，***RP***，***PC***，***RC*** 型を使えば，アプリケーションの一部を後で DLL に楽に変換できます。このマニュアルでは，ObjectWindows のプログラム例と段階別の STEP アプリケーション(¥BORLANDC¥OWL¥EXAMPLES ディレクトリ）でこれらの型を随所に使っていくことにします。

製品に関するお問い合わせ

ソフトウェアおよびマニュアルに関して，ご質問・ご意見などがありましたら，下記の**カスタマーサポートセンター**まで，お電話，書面またはファクシミリにてお問い合わせください。なお，お電話での受付時間は，平日の 10 : 00 am～12 : 00 am および 1 : 00 pm～5 : 00 pm とさせていただきます（土曜，日曜，祝日は休業とさせていただきますのでご了承ください)。

〒150　東京都渋谷区恵比寿西 2-6-11　川上ビル
ボーランド株式会社
カスタマーサポートセンター
(TEL) 03-5489-6671
(FAX) 03-5489-4873

お問い合わせの際には，下記の事項をお知らせください。

- 製品名とバージョン番号 (ObjectWindows for C++など)
- 製品のシリアル番号（マスターディスクに記載されているもの)
- コンピュータ名とモデル番号 (PC-9801 DA 21 など)
- オペレーティングシステムとバージョン番号 (NEC MS-DOS 3.30 C, MS-WINDOWS 3.0 など)

なお，書面にてお問い合わせの際には，CONFIG.SYS および AUTOEXEC.BAT の内容，簡潔なリストを添付されるようお願い致します。

また，標準以外のデバイスドライバや常駐プログラムなどが常駐している状態でおかしな動作をする場合には，それをはずした状態でもう一度確認されるようにお願いいたします。

参考書籍

Windows のプログラミングに関する情報は，Borland の言語コンパイラのマニュアルに掲載されています。Windows リソースのデザインと使用方法をさらに詳しく知りたい方には下記の図書をお薦めします。

Petzold, Charles. ***Programming Windows***, Microsoft Press. (Redmond, WA : 1990), 邦訳「**プログラミング WINDOWS**」アスキー出版局

Peter Norton and Paul Yao. ***PETER NORTON'S WINDOWS 3.0 POWER PROGRAMMING TECHNIQUES***, Bantam Books. 邦訳「**ピーター・ノートン WINDOWS 3.0 パワープログラミング上・下**」富士ソフトウェア株式会社

Systems Application Architecture Common User Access Advance Interface Design Guide, International Business Machines Corporation. (1989)

P A R T 1

第 1 部

ObjectWindowsとは

第1章

ウィンドウの継承

この章では，オブジェクト指向プログラミングに焦点を当てて，Microsoft Windowsのプログラミングを概説します。この例により，Windowsアプリケーションの内容とその結果を理解できるはずです。また，Windowsアプリケーションとしての必須条件や，ObjectWindowsを使ったオブジェクト指向プログラミングによるタスクの自動化および単純化についても理解できるように配慮しています。

Windowsアプリケーションとは

Windowsアプリケーションの主な構成要素を**図1.1**に示します。ObjectWindowsを使いこなすためにも，Windowsアプリケーションの各構成要素を知っておいてください。

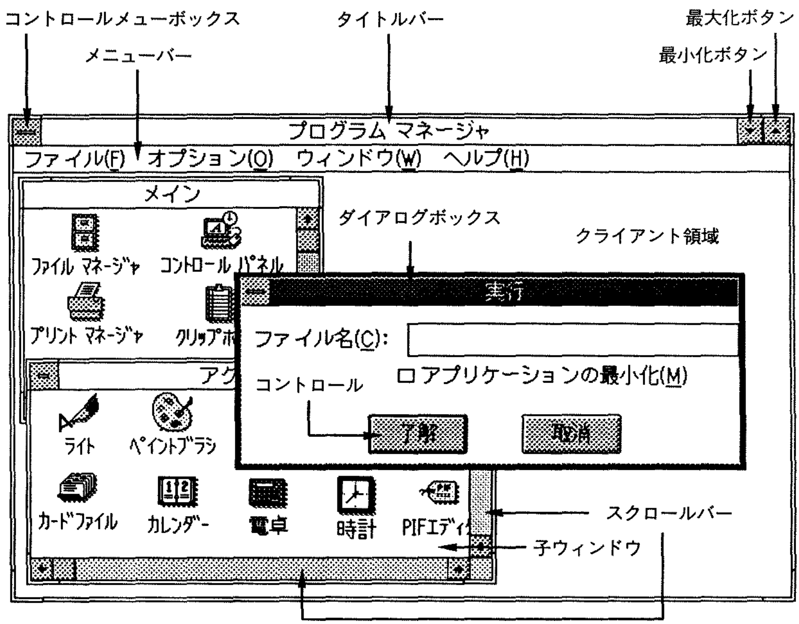


図 1.1
Windows アプリケーションの画面に表示される要素

Windows アプリケーションは特別なプログラムで，次の特徴を持っています。

- 特別な形式の実行（.EXE）ファイルである。
- Windows 環境下でのみ実行される。
- 通常，画面の矩形ウィンドウ内で実行される。
- 表示と実行の標準化のため，ユーザインターフェースガイドラインに従って作成される。
- 他の Windows アプリケーションおよび非 Windows アプリケーションと同時に実行できる。また，アプリケーション自身の他のインスタンスも同時に実行できる。
- 他の Windows アプリケーションとの通信およびデータの共有が可能である。

Windowsの有益性

Windowsは，ユーザと開発者の両者にとって有益です。ユーザにとっては次の点が有益です。

- 標準的で類推可能な操作体系。1つのWindowsアプリケーションの操作がわかれば，他のすべてのアプリケーションの操作が理解できます。
- アプリケーションごとのデバイスとドライバの設定が不要。Windowsは周辺機器をサポートするためのドライバを提供します。
- プロセス間のコンビネーション作業と通信。
- マルチタスクにより，複数のアプリケーションを同時に実行することができます。
- 大容量メモリ領域へのアクセス。Windowsは80286，80386，i486のプロテクトモードをサポートします。80386とi486ではバーチャルメモリをサポートします。

開発者にとっては次の点が有益です。

- デバイスに依存しないグラフィックス。グラフィックスアプリケーションが，すべての標準表示装置上（端末）で実行できます。
- 多種類のプリンタ，モニタ，マウスなどのポインティングデバイスをサポート。
- 豊富なグラフィックスのライブラリ。
- 大規模プログラムのための充分なメモリ確保。
- メニュー，アイコン，ビットマップなどのサポート。

ハードウェアの条件

このようにWindowsには多くの長所がありますが，反面ハードウェアの構成にいくつかの条件があります。通常，WindowsでDOSのアプリケーションと同等の効果を得るには高性能グラフィックス，大容量メモリ，高速プロセッサが必要です。Windowsの機能を十二分に引き出すには，80286以上のCPUを搭載したマシンと最低でも2MBのメモリが必要です。

オブジェクト指向のウィンドウ処理

ウィンドウ環境でのプログラミングでは，多くの新しい概念を理解しなくてはなりません。その結果，簡単なWindowsプログラムの開発であってもとても厄介なものになる恐れがあります。しかしObjectWindowsを使えばプロセスが単純になるので，アプリケーションの形式的な事項を意識せず，その機能だけに集中してプログラミングできます。

ObjectWindowsのフレームワークのおかげで，Windowsプログラムのかなり複雑な要素もオブジェクトを使って簡単に表わすことができます。ObjectWindowsのウィンドウオブジェクトはすべてのウィンドウに必要なデータをカプセル化して，共通のウィンドウ操作をこなし，共通のWindowsメッセージとイベントに応答します。ObjectWindowsのウィンドウクラスとアプリケーションクラスによりメッセージ処理が完全に管理されます。この処理は通常Windowsアプリケーションの大半を占めるものです。

Windowsに適したインターフェース

ObjectWindowsは，Borland C++のオブジェクト指向の機能を使って，Windowsのアプリケーションプログラムインターフェース（API）の各部をカプセル化しているため，プログラマはWindowsプログラミングの細かい点に気を配る必要がありません。したがって，オブジェクト指向でないやり方に比べて，少ない時間と労力でWindowsプログラムを作成できます。とりわけ，ObjectWindowsには次の3つの有用な機能があります。

■ ウィンドウ情報のカプセル化
■ Windows API 関数の抽象化
■ 自動メッセージ応答

ウィンドウ情報のカプセル化

ObjectWindows によって提供されるオブジェクトは，Windows アプリケーションのウィンドウ，ダイアログボックス，コントロールに対する動作とデータの格納場所を定義します。ObjectWindows アプリケーションでは**インターフェースオブジェクト**が，Windows の視覚的要素である**インターフェース要素**に相当します。インターフェースオブジェクトとインターフェース要素は相互に緊密な関係にありますが，この 2 つの違いを必ず認識しておいてください。

インターフェースオブジェクトを新しいインターフェース要素に対応させるにはそのメンバ関数 ***Create*** を間接的に呼び出します。***Create*** は該当する Windows 関数を呼び出して，新しいインターフェース要素を作成させます。Windows を呼び出したときに渡すインターフェースオブジェクトのデータメンバで，作成するインターフェース要素の物理的属性を定義します。Windows は作成したインターフェース要素のハンドルすなわち識別子を返します。インターフェースオブジェクトはこのハンドルをデータメンバ *HWindow* として格納します。

オブジェクトと要素の関係を成立させるのがウィンドウハンドルです。インターフェースオブジェクトを構築すると，インターフェース要素を作成するよう Windows に指示が出されます。Windows はこの要素を識別するハンドルを返し，オブジェクトがこれを *HWindow* というフィールドに格納します。多くの Windows 関数が，ウィンドウハンドルを引数にとるので，これをフィールドに格納しておけば，ウィンドウオブジェクトからいつでもアクセスできます。同様に，特定のウィンドウ用の描画ツールやステータス情報の格納場所としてインターフェースオブジェクトのフィールドを使用できます。

Windows関数の抽象化

Windowsアプリケーションは，Windows関数を呼び出して，アプリケーションの外観や動作を変化させることができます。関数は約600あり，Windowsアプリケーションプログラムインターフェース（API）を構成しています。それぞれの関数は多くの引数をとり，その種類も多様です。そこで，ObjectWindowsはこの関数呼び出しを抽象化するオブジェクトのメンバ関数を提供して，この呼び出しタスクを単純化しています。もちろん，Borland C++から直接Windows関数を呼び出すこともできます。

前にも述べたように，Windows関数の多くの引数はインターフェースオブジェクトのデータメンバに格納されています。そのため，メンバ関数ではこのデータを使ってWindows関数に引数を指定するという方法が可能になります。さらに，ObjectWindowsでは，関連する関数の呼び出しをまとめてメンバ関数とし，各メンバ関数で高レベルのタスクを実行するようにしています。この結果，Windowsとのインターフェースがすっきりとして使いやすくなりました。

このメンバ関数を使うと，何百もあるWindows API関数を個別に操作しないですみます。また，APIを直接呼び出すこともできます。ObjectWindowsは両者の長所，すなわち高レベルのオブジェクト指向の開発環境とグラフィックス環境の最大限のコントロールを提供しています。

メッセージ応答の自動化

ほとんどのアプリケーションは，Windows環境に処理を指示するばかりでなく，マウスのクリックなどのユーザの行為，他のアプリケーション，あるいは他のソースから来る何百ものWindowsメッセージに応答できなくてはなりません。ユーザがマウスのボタンをクリックするか，マウスを移動させると，そのつどWindowsはアプリケーションにメッセージを送ります。したがって，アプリケーションは数分間のうちに数千ものメッセージを受け取ることもあります。メッセージを正しく処理して応答することは，アプリケーションを正常に動作させるための必須条件です。

オブジェクトはメンバ関数を継承し，定義し直すことができるので，Windowsのメッセージに対して間違いなく応答することができます。ObjectWindowsはWindowsメッセージを受け取り，これをBorland C++のメンバ関数呼び出しに変換します。したがって，ObjectWindowsを使えば，アプリケーションが処理しなければならない各メッセージに対する応答メンバ関数だけを定義しておけばよいのです。たとえば，ユーザがマウスの左ボタンを押すと，Windowsは ***WM_LBUTTONDOWN*** メッセージを発行します。このようなマウスクリックに応答するウィンドウやコントロールをプログラム中に記述したい場合には，***WM_LBUTTONDOWN*** メッセージに対応する ***WMLButtonDown*** メンバ関数を定義します。そうすると，Windowsからこのメッセージが送られてくるたびに，オブジェクトは定義したメンバ関数を自動的に呼び出します。

このようなメンバ関数を**メッセージ応答関数**と呼びます。ダイナミックディスパッチバーチャルテーブル（DDVT）というBorlandのC++コンパイラ機能を使うと，特定のWindowsメッセージをメッセージ応答関数に対応づけることができます。**ダイナミックディスパッチメンバ関数**を宣言するときは整数型**ディスパッチインデックス**も指定してください。たとえば以下の例の ***Test*** は，クラスのディスパッチインデックスが100のダイナミックディスパッチメンバ関数として定義されます。

```
virtual void Test() = [100];
```

WINDOWS.Hを見ればわかりますが，Windowsメッセージには ***WM_LBUTTONUP*** のように記述的な名前が与えられていますが，実は単に整数値にすぎません。そのため，WindowsメッセージでDDVTを使えるというわけです。

ObjectWindowsアプリケーションからのWindowsメッセージは，ディスパッチインデックスがメッセージの値と等しい関数に送られます。たとえば，以下の例はWindowsメッセージの ***WM_FIRST*** + ***WM_LBUTTONUP*** に応答するvoid関数を宣言するものです。

```
virtual void WMLButtonUp(RTMessage Msg) = [WM_FIRST + WM_LBUTTONUP];
```

定数WM_FIRSTとWM_LBUTTONUPの和は整数値であり，Windowsはこれでメッセージを識別します。

これ以上のことはする必要がありません。というのは，ObjectWindowsは新しいメッセージ応答関数が追加されるとこれを認識し，自動的にメッセージをこのメッセージ応答関数に送るからです。ObjectWindowsが認識できるように，メッセージ応答関数の引数には ***RTMessage*** 型を使います。前にも述べましたが，***RTMessage*** は ***TMessage*** への参照です。***TMessage*** オブジェクトには，Windowsがメッセージで渡すすべての情報が入っています。

☞ ダイナミックディスパッチメンバ関数は使用上の制約がいくつかあります。以下に説明します。

- 基本クラスのダイナミックディスパッチ関数を派生クラスで定義し直す場合は，同じディスパッチインデックスを使います。次の例は無効です。

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase
{
  virtual void sample() = [299];
};
```

ディスパッチインデックスが異なるので，この定義は無効です。

- クラス階層内のダイナミックディスパッチ関数には固有のディスパッチインデックスが必要です。次の例は無効です。

```
class TestBase
{
  virtual void sample1() = [100];
  virtual void sample2() = [200];
  virtual void sample3() = [100];
};
```

同じディスパッチインデックスに2つの異なる関数を割り当てようとしているので，この文は無効です。

- 基本クラス内の「普通の」(すなわちダイナミックディスパッチでない) 仮想関数を，派生クラスでダイナミックディスパッチ関数によって定義し直すことはできません。次の例は無効です。

```
class TestBase
{
  virtual void sample();
};

class TestDerived : public TestBase
{
  virtual void sample() = [100];
};
```

ダイナミックディスパッチ関数で普通の関数を定義し直そうとしているので，この定義は無効です。

- 基本クラス内のダイナミックディスパッチ関数を，派生クラスで「普通の」仮想関数によって定義し直すことはできません。次の例は無効です。

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase
{
  virtual void sample();
};
```

普通の仮想関数でダイナミックディスパッチ関数を定義し直そうとしているので，この定義は無効です。

■ クラスが複数の基本クラスから派生している場合，その派生クラスは多重継承していると言います。ダイナミックディスパッチ関数を持つことができるのは，多重継承している派生クラスの基本クラスのうち最初の基本クラスだけです。たとえば次のクラス定義は有効です。

```
class A
{
  virtual void AA() = [100];
};


class B
{
};

class C : public A, public B
{
};
```

この例では，ダイナミックディスパッチ関数が最初の基本クラスにあるので有効です。

■ 次の例は無効です。

```
class A
{
};

class B
{
  virtual void BB() = [100];
};

class C :  public A, public B
{
};
```

2番目の基本クラスであるクラスBにダイナミックディスパッチメンバ関数があるので，この定義は無効です。

■クラスオブジェクトをポインタが指し，特定のスコープのオーバーライドが使われていない場合は，ダイナミックディスパッチ関数を呼び出せません。次の例は無効です。

```
class Base
{
  virtual void sample() = [100];
};

int main(void)
{
  Base b;
  Base *BasePtr = &b;

  BasePtr->sample();
}
```

ポインタを通じてダイナミックディスパッチ関数を直接呼び出すことができないので，この文は無効です。

■仮想基本クラスはダイナミックディスパッチメンバ関数を持てません。次の例は無効です。

```
class Base
{
  virtual void sample() = [100];
};

class Derived : virtual public Base
{
};
```

仮想基本クラスはダイナミックディスパッチメンバ関数を持てないので，この定義は無効です。

Windowsメッセージに応答する別の方法として，プログラムの応答が必要な各Windowsメッセージに対し **case** つきの長い **switch** 文を使う方法があります。この方法は，Microsoft Cプログラミングには必要ですがObjectWindowsでは採用していません。次の例は，Cで書いたWindowsアプリケーションの一部ですが，**switch** 文の中に **switch** 文が入っています。

このコード例は Windows メッセージに対する非オブジェクト指向の応答方法を示しています。

```
switch (message)
{
  case WM_DDE_CLOSED:
    CloseMsgWnd();
    break;

  case WM_COMMAND:
    switch (wParam)
    {
      case IDM_QUIT:
        //ユーザがメニューのQUITを選択
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        break;

      case IDM_HOME:
        // GRPHキーを押下
        SendMessage(hWnd, WM_HSCROLL, SB_TOP, 0L);
        SendMessage(hWnd, WM_VSCROLL, SB_TOP, 0L);
        break;

      case IDM_ABOUT:
        // ユーザがメニューのABOUTを選択
        lpproc = MakeProcInstance(About, hInst);
        DialogBox(hInst, MAKEINTRESOURCE(ABOUT), hWnd, lpproc);
        FreeProcInstance(lpproc);
        break;
        ⋮
```

約 200 個もある Windows メッセージを考慮すると，この **switch** 文を使う方法ではすぐに収拾がつかなくなるのは明らかです。ObjectWindows の DDVT を使う方が簡単です。

Windows プログラムの構造

DOS, Windows, 独自のアプリケーション，その他のアプリケーションといった多くのソフトウェア要素は同時に作用し合うので，Windows アプリケーションの各部と他の外部要素との関係を知っておくとプログラミングの助けになります。この節では，Windows の構造と，Borland C++で ObjectWindows を使って記述した代表的な Windows アプリケーションについて説明します。

Windows の構造

実行時，Windows と API の各機能は，次の 3 つの外部ライブラリモジュールに存在しており，実行中のアプリケーションによって呼び出されます。

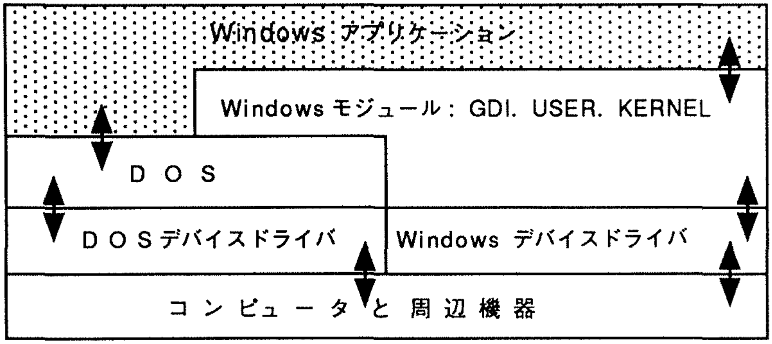
- KERNEL.EXE—メモリとリソースの管理，スケジュール管理，DOS とのやり取りを統括します。
- GDI.EXE—グラフィックスイメージの画面への表示，プリンタへの出力を統括します。
- USER.EXE—ウィンドウ管理，ユーザ入力，通信を統括します。

これらのモジュールは市販の Windows の一部であるため，Windows がインストールされていれば立ち上げ時にロードされています。これらのライブラリモジュールを利用するプログラムを作成することはできますが，ライブラリモジュールを実際にインクルードすることはできません。

Windows および DOS との対話

DOS の有効範囲は限定されているので，DOS アプリケーションの実行に関与する DOS の役割は見落されがちです。それでも，アプリケーションコードとオペレーティングシステム（OS）の諸機能との間に対応関係があるために，DOS プログラムが実行できるのです。同様のことが Windows プログラムにもあてはまります。Windows は非常に多くの OS 関数を提供しているため，Windows とアプリケーション間の相互関係を見落とすことはあまりありません。アプリケーションプログラムは絶えずオペレーティングシステム（DOS と Windows）と対話するように構成されていなければなりません。

図1.3
Windowsアプリケーション，Windows，DOSの連携（網かけの部分がユーザアプリケーション側で記述すべき部分）



“Hello, World”

新しい言語や環境を紹介する場合，その言語や環境の下で“Hello, World”プログラムを作って説明するのが従来からの方法なので，このマニュアルでもこの方法をとります。通常，このプログラムは，“Hello, World”という文字列だけを画面に表示するコードだけです。

もちろん，Windowsではもっと複雑です。ウィンドウを表示し，ここにデータを書き込み，このウィンドウに外部環境との対話方法を指示しなければなりません。少なくともウィンドウをクローズして終了させるための対話が必要です。このような簡単な処理を行なわせるにも，設計をゼロからスタートするとかなり手間がかかります。

Windowsで実行されるアプリケーションには多数の必須条件が課せられるので，単純なプログラムでもかなりの行数になってしまいます。しかし，ObjectWindowsで記述したプログラムは，メインウィンドウの作成や表示などほとんどの必須条件を自動的に満足するようになっています。このおかげで，“Hello, World”プログラムは次のように簡単に記述できます。

HELLOAPP.CPP です。

```
#include <owl.h>

// TApplication からの派生クラスを定義
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
            hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

// THelloApp の MainWindow データメンバを構築
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance,
                      lpCmdLine, nCmdShow);
   HelloApp.Run();
   return HelloApp.Status;
}
```

アプリケーション起動時の処理

アプリケーションが実行を開始すると，Windows から 4 つの引数が渡されます。ObjectWindows プログラムでは，この 4 つの引数を，***TApplication*** から派生したアプリケーションオブジェクトのコンストラクタに渡します。この引数はアプリケーションオブジェクトのデータメンバにも格納されるので，プログラムからでも簡単にアクセスできます。

次のアプリケーションデータメンバに格納されます。

- ***hInstance***——アプリケーションのインスタンスを指すハンドルを保持します。
- ***hPrevInstance***——同じアプリケーションの最後のインスタンスを指すハンドルを保持します。最初のインスタンスへのハンドルの場合はゼロになります。
- ***lpCmdLine***——ファイル名オプションなど，アプリケーションのスタートアップ時にコマンドラインで記述した文字列を ***LPSTR*** で保持します。たとえば，‘CALC.EXE/M’ や ‘WORDPROC.EXE LETTER 1.DOC’ などです。

■ ***nCmdShow***――メインウィンドウの最初のモードを表わす整数を格納します。***Show*** メンバ関数を呼び出すときに使います。

ObjectWindowsが最初にすべきことは，基本クラス ***TApplication*** からアプリケーションオブジェクトを派生させ構築することです。*THelloApp* の例で説明します。

ObjectWindowsアプリケーションの最初のインスタンスの場合，***Run*** は ***InitApplication*** を呼び出して，アプリケーション全体に対して必要な初期化を行なわせます。その後，***InitInstance*** を呼び出し，最初のインスタンスを含むすべてのインスタンスを初期化させます。***InitInstance*** はメンバ関数 ***InitMainWindow*** を呼び出し，メインウィンドウオブジェクトの構築と初期化を行わせます。

最後に ***Run*** は，ObjectWindowsアプリケーションのメッセージループを起動します。プログラムが終了するまで ***Run*** は値を返しません。アプリケーションの終了状態はデータメンバ ***Status*** に格納されています。この値は前に説明した ***WinMain*** の場合と同様，Windowsに返してください。

メインウィンドウの役割

アプリケーションのメインウィンドウとは，アプリケーションを起動すると最初に現われるウィンドウのことです。このウィンドウには，コマンド一覧がメニュー形式で表示されます。アプリケーション実行中，メインウィンドウはアプリケーションのインターフェースを管理します。また多くの場合，プログラムの唯一の作業領域として使われます。複雑なアプリケーションでは複数のウィンドウが作業領域として使われます。最後に，ユーザがメインウィンドウをクローズすると，このメインウィンドウがアプリケーションをクローズするためのプロセスを起動します。

アプリケーション開発サイクル

Windows アプリケーションには，メインウィンドウの初期化など，いくつかの必須条件があるので，通常，既存の Windows アプリケーションを利用し，これをカスタマイズするのが一番の近道です。ObjectWindows は，多くのサンプルプログラムを提供しているので，自分のアプリケーションに一番合ったものを選んでください。次のような一般的なステップでプログラミングを行なうことができます。

1. プログラムコードの作成。
2. ダイアログボックス，メニューなどのリソースの作成。
3. プログラムのコンパイル。
4. プログラムのデバッグ。

第2章

Windowsを使ってみよう

ObjectWindowsライブラリの概略がわかったところで，簡単なObjectWindowsプログラムを構築してみましょう。以降の章で，グラフィックスを使った対話型のWindowsプログラムを構築することにします。メニュー，ファイルのセーブとロード，グラフィックスとテキストの描画，さらに簡単なヘルプシステムもあるプログラムを構築します。サンプルプログラムを構築していくうちに，メッセージの処理，ウィンドウの親子関係の管理，グラフィックスの自動再描画といった，Windowsアプリケーションの設計上の基本原理が理解できるはずです。

プログラムの作成演習は，次の10段階からなり，第2章から第6章で解説します。

- ステップ1：簡単なWindowsアプリケーション
- ステップ2：メインウィンドウクラス
- ステップ3：ウィンドウにテキストを描画する
- ステップ4：ウィンドウに線を描画する
- ステップ5：線の太さを変更する
- ステップ6：グラフィックスを描画する
- ステップ7：メインウィンドウのメニュー
- ステップ8：ポップアップウィンドウを追加する
- ステップ9：描画情報をファイルに格納する
- ステップ10：ヘルプウィンドウをポップアップとして表示する

デフォルトのインストール用ディレクトリを変えていなければ，メインディレクトリOWLのサブディレクトリEXAMPLES¥STEPSにソースコードがあります。ファイル名は，チュートリアルの各ステップに対応して，STEP1.CPP，STEP2.CPP，… になっています。

図 2.1 はこのチュートリアルの最後にできあがるアプリケーションです。

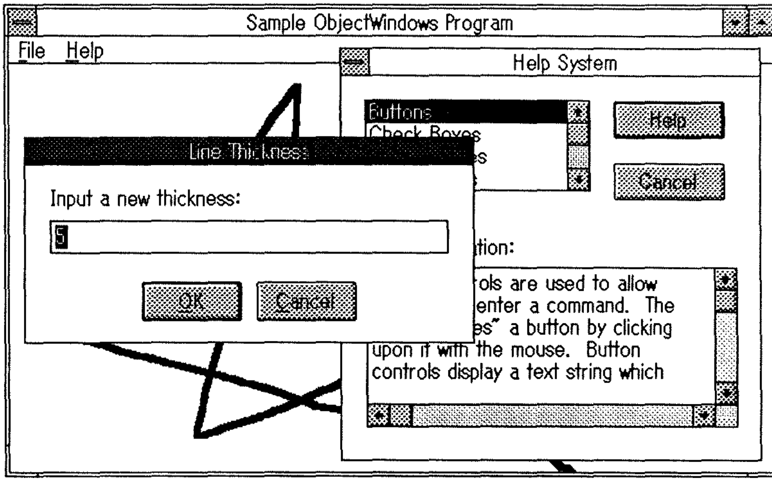


図 2.1
完成した ObjectWindows アプリケーション

ObjectWindows アプリケーションの構築：予備知識

ObjectWindows アプリケーションの構築を始める前にやるべきことがいくつかあります。まず，コンパイラとリソースコンパイラが該当ファイルを見つけられるように，インクルードディレクトリを正しく指定しておいてください。次に，TLINK がアプリケーションに必要なすべてのライブラリを見つけられるように，ライブラリのディレクトリを正しく指定しておきます。最後に，該当する ObjectWindows リソースを入れたリソースファイルを作成し，これをアプリケーションに結合しておきます。

コンテナクラスライブラリ

ObjectWindows のクラスライブラリは，サブディレクトリ CLASSLIB にあるコンテナクラスライブラリにより決まります。ObjectWindows のオブジェク

トはすべてコンテナクラスライブラリで定義されている **Object** を基本クラスとして共有します。その他にもコンテナクラスライブラリのクラスには ObjectWindows クラスで使われるものがあります。コンテナクラスはどれでも ObjectWindows プログラムで使えます。

ディレクトリ

アプリケーションに必要な STDIO.H などの C++の標準ヘッダファイル以外にも，OWL.H などの ObjectWindows ヘッダファイルがどこにあるかを各種のコンパイラツールに指示しなければなりません。OWL.H なしでは，どんな簡単な ObjectWindows プログラムも記述できません。アプリケーションが複雑になると，他のヘッダファイルも必要になります。たとえば，***TButton*** クラスを使うプログラムには，BUTTON.H が必要です。ObjectWindows アプリケーションにも ObjectWindows 自体にも，ObjectWindows でコンテナクラスライブラリを指定してやらなければなりません。したがって，ObjectWindows コンテナクラスライブラリのインクルードディレクトリのディレクトリパスも含めることを忘れてはいけません。

どのアプリケーションも，Borland C++の標準ライブラリばかりでなく，作成したライブラリや購入したライブラリを使えるようにしなければなりません。ObjectWindows アプリケーションの場合はさらに，ObjectWindows が指定したコンテナクラスライブラリと ObjectWindows の標準ライブラリも使えなければなりません。

次の表は ObjectWindows または Borland C++のインストール用ユーティリティが使うデフォルトのディレクトリをまとめたものです。インストール用ユーティリティに対し別のディレクトリを指示した場合は，メイクファイルとプロジェクトファイルの指定を変更して実際の設定に合わせてください。

表 2.1　デフォルトのインクルードディレクトリ

種類	デフォルトのインクルードディレクトリ	デフォルトのライブラリディレクトリ
ObjectWindows	¥BORLANDC¥OWL¥INCLUDE	¥BORLANDC¥OWL¥LIB
コンテナクラスライブラリ	¥BORLANDC¥CLASSLIB¥INCLUDE	¥BORLANDC¥CLASSLIB¥LIB
Borland C++	¥BORLANDC¥INCLUDE	¥BORLANDC¥LIB

正しいライブラリの指定

ObjectWindows のライブラリとコンテナクラスライブラリはそれぞれ 3 種類あり，ラージ，ミディアム，スモールの各種メモリモデルに対応しています。これらのライブラリは次の表に示すライブラリディレクトリに収容されています。

表 2.2 各メモリモデルのライブラリ

メモリモデル	ObjectWindows ライブラリ	コンテナクラスライブラリ
スモール	OWLWS.LIB	TCLASSWS.LIB
ミディアム	OWLWM.LIB	TCLASSWM.LIB
ラージ	OWLWL.LIB	TCLASSWL.LIB

DLL を使う ObjectWindows アプリケーション

ObjectWindows と Borland C++のランタイムライブラリとして，次の表に示す DLL（ダイナミックリンクライブラリ）とインポートライブラリを使えます。アプリケーションにどちらかの DLL を使わせるときは，対応するインポートライブラリをリンクさせなければなりません。

表 2.3 DLL とインポートライブラリ

種類	DLL	インポートライブラリ
ObjectWindows	OWL.DLL	OWL.LIB
Borland C++	BCRTL.DLL	CRTLL.LIB

どちらかの DLL だけを単独で使うことはできません。つまり，ObjectWindows の DLL を使うアプリケーションでは，Borland C++のランタイムライブラリの DLL も使います。この逆の場合も同様です。DLL を使う場合はラージメモリモデル（コマンドラインコンパイラの **-ml** か IDE のメニューオプション Options | Compiler | Code Generation）とスマートコールバックオプション（コマンドラインコンパイラの **-WS** か IDE のメニューオプション Options | Compiler | Entry/Exit Code）を使ってください。また，コンテナクラスライブラリは ObjectWindows の DLL (OWL.DLL) についているので，ObjectWindows DLL を使うときは，コンテナクラスライブラリを明示的にリンクする必要はありません。

リソースファイルの作成

リソースファイルの構築にはリソースエディタを使うのが普通ですが，コマンドラインのリソースコンパイラ RC も使えます。ObjectWindows のリソースファイルは，Windows のリソースファイルと似ています。ObjectWindows 専用のインクルードファイルを追加する場合もあります。ObjectWindows には，.DLG または.RC という拡張子を持つリソースファイルがあります。これらのリソースファイルのリソースを使う ObjectWindows のクラスは次の 4 つです。

表 2.4
リソースファイルの必要な ObjectWindows クラス

クラス	必要なリソースファイル
TInputDialog	INPUTDIA.DLG
TFileDialog	FILEDIA.DLG
TEditWindow	STDWNDS.DLG, EDITACC.RC, EDITMENU.RC
TFileWindow	STDWNDS.DLG, FILEACC.RC, FILEMENU.RC

ObjectWindows アプリケーションでこれらのクラスを使う場合は，該当する ObjectWindows のリソースファイルがアプリケーションのリソースファイルにインクルードされていることを確認してください。***TInputDialog*** と ***TFileDialog*** を使ったアプリケーションのリソースファイル例を以下に示します。

```
#include <windows.h>
#include <owlrc.h>
rcinclude inputdia.dlg
rcinclude filedial.dlg
⋮
```

アプリケーションに必要な他のリソースは，前に述べたインクルード指令に従います。ObjectWindows で提供されるリソースファイルと OWLRC.H は，ObjectWindows のインクルードディレクトリにあります。このディレクトリは RC に対するコマンドラインオプションとして指定してください。WINDOWS.H は，Borland C++インクルードディレクトリにあるので，これも RC に対するコマンドラインオプションとして指定してください。インクルードディレクトリの指定には，RC のコマンドラインオプション **-i** を使います。**-i** オプションを繰り返せば，複数のディレクトリを指定できます。

ObjectWindows アプリケーションの構築：詳細

ObjectWindows アプリケーションの構築の予備知識を習得できたので，いよいよ詳細に入りましょう。

ObjectWindows アプリケーションは，3 つの方法で構築できます。最初の最も簡単な方法は，提供されたメイクファイルの 1 つで対象を変更することです。2 番目は，Borland C++の統合開発環境（IDE：Integrated Development Environment）のプロジェクトマネージャを使う方法です。最後の方法では，コマンドラインツールを直接使います。メイクファイルを使うのもコマンドラインツールを直接使うのも非常に似ているので，ここでは 2 番目と 3 番目の方法を説明します。

Borland C++のツールである IDE とコマンドラインツールに精通した読者を対象として説明します。また，一般的な Windows アプリケーションの構築方法にも詳しいことを前提にします。詳細については Borland C++のマニュアル「**イントロダクション**」を参照してください。

IDE で ObjectWindows アプリケーションを構築する場合

Borland C++の IDE で ObjectWindows アプリケーションを構築するには，**プロジェクト**ファイルが必要です。プロジェクトの使い方については，IDE の関連マニュアルを参照してください。ObjectWindows アプリケーションのプロジェクトを構築するときの一般的な手順は以下のとおりです。

- 多数の（たいていは大型の）ヘッダファイルのコンパイルを迅速にするため，プリコンパイルヘッダオプションを選択する。
- プロジェクトファイルをオープンする。
- Windows のアプリケーション型 EXE を選択する。
- アプリケーションと併用して，ObjectWindows の DLL，Borland C++ランタイムライブラリの DLL，ユーザ DLL のいずれかを使う場合は，スマートコールバックオプションとラージメモリモデルを選択してください。
- 他に必要なコンパイラとリンカオプションを選択します。
- インクルードディレクトリを指定します。デフォルトのインクルードディレクトリを以下に示します。

ディレクトリの詳細については，31 ページを参照してください。

・¥BORLANDC¥OWL¥INCLUDE
・¥BORLANDC¥CLASSLIB¥INCLUDE
・¥BORLANDC¥INCLUDE

■ライブラリディレクトリを指定します。デフォルトのライブラリディレクトリを以下に示します。

・¥BORLANDC¥OWL¥LIB
・¥BORLANDC¥CLASSLIB¥LIB
・¥BORLANDC¥LIB

警告！ヘッダファイル.H は追加しないでください。

■アプリケーションのソースファイル（.CPP，.C，.ASM などの拡張子を持つファイル）をプロジェクトに追加します。

警告！ダイアログリソースファイル.DLG は追加しないでください。

■アプリケーションのリソースファイル.RC をプロジェクトに追加します。詳細については，リソースファイルに関する前の節を参照してください。以降の例では，リソースファイル STEPS.RC を追加します。

ライブラリの詳細については 32 ページを参照してください。

■ObjectWindows をアプリケーションに静的にリンクさせておく場合は，使用するメモリモデルに合った ObjectWindows ライブラリを追加します。たとえば，スモールモデルのアプリケーションには OWLWS.LIB を追加します。ObjectWindows の DLL（OWL.DLL）を使いたければ，ObjectWindows の DLL に対するインポートライブラリ OWL.LIB を追加します。

■アプリケーションのメモリモデルに合った ObjectWindows コンテナクラスライブラリを追加します。たとえば，ミディアムモデルのアプリケーションには TCLASSWM.LIB を追加します。ObjectWindows の DLL を使う場合，コンテナクラスライブラリを明示的にリンクする必要はありません。

■ObjectWindows の DLL を使う場合，必ず Borland C++ のランタイムライブラリをアプリケーションにリンクしてください。こうすると，.EXE ファイルのサイズを小さくできます。インポートライブラリ CRTLL.LIB をプロジェクトに追加してください。

■ObjectWindows のライブラリ（OWLWx.LIB）を静的にリンクする場合，Windows の新しいグローバルメモリサブアロケータライブラリも静的にリンクしてください。こうすると.EXE ファイルのサイズを小さくできます。SALLOCWx.LIB（"x" はメモリモデルを示す）をプロジェクトに追加してください。

■アプリケーションを構築します。IDE のプロジェクトマネージャがプロジェクトを調べて，アプリケーションのソースファイルとリソースファイル.RC のコンパイルが必要かどうかを判断します。次に，リンカが中間のオブジェクトファイル.OBJ とライブラリをリンクします。最後に.EXE ファイルまたは

.DLL ファイルにリソースが追加されます。

コマンドラインツールで ObjectWindows アプリケーションを構築する場合

Borland C++コマンドラインコンパイラで ObjectWindows アプリケーションのコンパイルとリンクを行なう場合は，コンフィギュレーションファイルを編集して（なければ作成して），インクルードディレクトリとライブラリディレクトリを指定するのが最も簡単な方法です。コンフィギュレーションファイルで **-I** 指令か **-L** 指令を使って，該当するディレクトリを指定してください。

詳細については 31 ページのインクルードディレクトリとライブラリディレクトリの説明を参照してください。

アプリケーションをコンパイルしてリンクするには，次のようなコマンドラインを使います。

```
BCCX -WE myprog.cpp owlws.lib sallocws.lib tclassws.lib
```

このコマンドは MYPROG.CPP をコンパイルしてから，これを OWLWS.LIB と TCLASSWS.LIB にリンクします。

☞ ObjectWindows の DLL，Borland C++ランタイムライブラリの DLL，ObjectWindows のユーザ DLL のいずれかをアプリケーションと併用する場合は，コマンドラインオプション **-WS**（スマートコールバック）でコンパイルしてください。なお，ObjectWindows の DLL を使う場合は，次の例のように，ラージメモリモデルでアプリケーションを構築することを忘れないでください。

```
BCCX -WS -ml myprog.cpp owl.lib crtll.lib
```

このコマンドは MYPROG.CPP をコンパイルしてから，これをインポートライブラリ OWL.LIB と CRTLL.LIB，およびスタティックライブラリ TCLASSDL.LIB にリンクします。

次に，アプリケーションのリソースファイルを以下のコマンドラインでコンパイルします。

```
RC -i ¥borlandc¥owl¥include -i ¥borlandc¥include myprog.rc
myprog.exe
```

このように，RC のコマンドラインでインクルードディレクトリを明示的に指定しない場合は，INCLUDE という環境変数を作成して，次のように RC にインクルードディレクトリを指定してください。

```
SET INCLUDE=¥owl¥include;¥borlandc¥include
RC myprog.rc myprog.exe
```

SET 文は AUTOEXEC.BAT ファイルに入れてもかまいません。

どちらの場合でも，RC はリソースファイル.RC をバイナリのリソースファイル.RES にコンパイルしてから，.RES ファイルをアプリケーションの.EXE ファイルに付加します。これで，実行形式の ObjectWindows アプリケーションになります。

ステップ 1：簡単な Windows アプリケーション

まず，STEP アプリケーションという ObjectWindows アプリケーションの骨組みから記述します。このプログラムは STEP1.CPP にあり，ObjectWindows を利用して記述するすべてのプログラムの出発点になります。ステップ 1 で STEP アプリケーションはアプリケーションのメインウィンドウのインスタンスを構築し作成します。

アプリケーションの必須条件

Windows プログラムにはすべてメインウィンドウがあり，ユーザがプログラムを起動するとオープンされます。ユーザがメインウィンドウをクローズするとアプリケーションが終了します。ObjectWindows アプリケーションでは，通常メインウィンドウを**ウィンドウオブジェクト**として扱います。このオブジェクトは**アプリケーションオブジェクト**によって**所有**されます。アプリケーションオブジェクトは，メインウィンドウを作成して表示し，Windows メッセージを処理し，さらにアプリケーションを終了させます。アプリケーションオブジェクトはアプリケーションのかわりにオブジェクト指向の処理を行なうものです。同様に，ObjectWindows はウィンドウ，ダイアログなどのクラスを提供して Windows プログラミングの詳細を意識させない仕組みになっています。

それぞれのObjectWindowsプログラムでは，提供されているクラス ***TApplication*** から新しいアプリケーションクラスを派生させて定義しなければなりません。Windowsプログラムのエントリポイントである *WinMain* で派生クラス（**アプリケーションオブジェクト**）のインスタンスが構築されます。型（クラスとそのインスタンス）の名前は通常 *T* で始まり，型を指すポインタは *PT* で始まることになっています。STEPアプリケーションでは，このクラスを *TMyApp* と呼びます。

STEPアプリケーションの主な機能を以下に示します。

STEP 1.CPPの一部です

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance,lpCmdLine, nCmdShow);
   MyApp.Run();
   return MyApp.Status;
}
```

最初の文で *WinMain* はSTEPアプリケーションの**アプリケーションオブジェクト**を構築します。“Sample ObjectWindows Program”がコンストラクタに渡され，データメンバ ***Name*** の値になります。*WinMain* に渡された他の引数もコンストラクタに渡され，このアプリケーションオブジェクトのデータメンバに格納されます。次に *MyApp.Run* が呼び出され，ObjectWindowsアプリケーションの動作を開始させます。最後に最終状態（アプリケーションオブジェクトのデータメンバ ***Status***）が返されます。

アプリケーションクラスの定義

アプリケーションは，ObjectWindowsの標準クラス ***TApplication***（あるいは ***TApplication*** から派生したクラス）から新しいクラスを派生させなければなりません。この新しいクラスでは，少なくともメンバ関数 ***InitMainWindow*** のオーバーライドが必要です。***InitMainWindow*** は，ObjectWindowsアプリケーションを初期化するときにメインウィンドウオブジェクトを構築します。クラス *TMyApp* は次のように定義します。

STEP 1.CPP の一部です。

```
class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
   virtual void InitMainWindow();
};
```

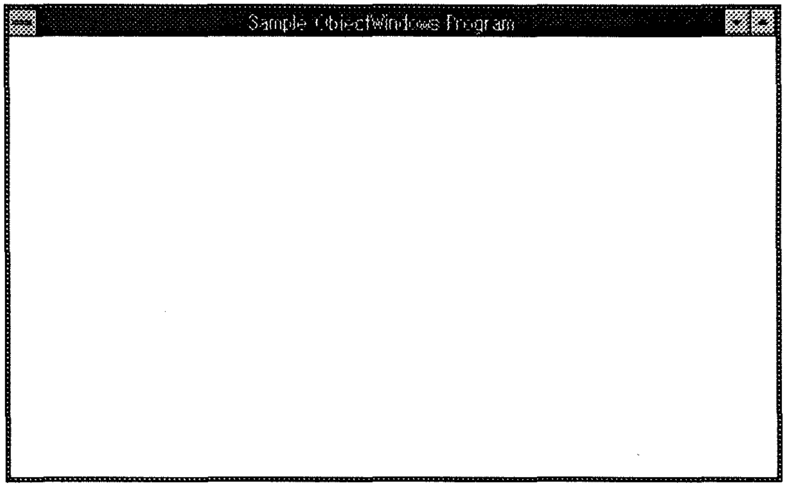
このメインウィンドウオブジェクトは，アプリケーションオブジェクトのデータメンバ ***MainWindow*** に格納されます。アプリケーションオブジェクトがメインウィンドウオブジェクトを**所有**すると言いますが，両者の間に階層関係はありません。このような所有関係を**インスタンスリンケージ**と呼びます。*TMyApp* : : *InitMainWindow* は次のように定義します。

STEP 1.CPP の一部です。

```
void TMyApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, Name);
}
```

TMyApp のメインウィンドウオブジェクトは上のように ***TWindow*** クラスのインスタンスとして構築されます。渡された引数 NULL は，このウィンドウがアプリケーションウィンドウのメインウィンドウになることを示します。2 番目の引数はウィンドウのタイトルです。アプリケーションのデータメンバ ***Name*** を渡します。このデータメンバには "Sample ObjectWindows Program" が設定してあります。アプリケーションのメインウィンドウは通常，ObjectWindows クラスから派生させたクラスのインスタンスとして構築されます。ここで構築した汎用ウィンドウを，ステップ 2 でもっと実用的な派生クラスのウィンドウと置き換えてみましょう。**図 2.2** は STEP アプリケーションの外観です。

図 2.2
最初の ObjectWindows プログラム STEP アプリケーション



この時点では，STEP アプリケーションは空のウィンドウを表示するだけですが，このウィンドウは移動，サイズ変更，最大化，最小化，クローズが可能です。これまでの STEP アプリケーション全体のリストを次に示しておきます。

STEP 1.CPP の内容

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

void TMyApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);
   MyApp.Run();
   return MyApp.Status;
}
```

ステップ2：メインウィンドウクラス

ステップ1で作成したプログラムは，アプリケーションオブジェクトとウィンドウオブジェクトという2つのオブジェクトでできています。アプリケーションオブジェクト *MyApp* は，***TApplication*** から派生させたクラス *TMyApp* のインスタンスです。ウィンドウオブジェクトのポインタは，アプリケーションオブジェクトのデータメンバ ***MainWindow*** に保持されています。これは，ObjectWindows の汎用ウィンドウである ***TWindow*** のインスタンスを指すポインタです。メインウィンドウ用の独自のウィンドウクラスを定義して，アプリケーション固有の動作を組み込みます。この節では，***TWindow*** から派生させた専用のウィンドウクラスを定義して，メインウィンドウオブジェクトを実現することにします。

ウィンドウオブジェクトとは

ステップ1では，メインウィンドウの構築など，Windows アプリケーションの標準的動作がアプリケーションオブジェクトにカプセル化されることを学びました。クラス ***TApplication*** が ObjectWindows アプリケーションの基本的動作を提供します。

同様に，各ウィンドウに必要な動作は各ウィンドウオブジェクトにカプセル化されます。ウィンドウには次の動作が必要です。

- ウィンドウ自体の表示とサイズ変更
- イベントへの応答
- 子ウィンドウの管理
- 規定どおりの順序でのクローズ

ObjectWindows のウィンドウクラスで上の動作が提供されます。プログラムの利用価値を高め，魅力あるものにするには，提供されている ObjectWindows クラスから新しいウィンドウクラスを派生させ作成する必要があります。新しいクラスはメンバ関数とデータメンバを継承し，さらに独自のメンバ関数とデータメンバを追加します。すなわち，オブジェクト指向の手法を使えば，そのつどウィンドウを作り直す手間が省けるのです。

ハンドル

どのウィンドウオブジェクトにも，***HWindow***，***Parent***，**子リスト**の3つのデータメンバがあります。***HWindow*** にはウィンドウを指すハンドルが入ります。ハンドルはウィンドウ，ダイアログボックス，コントロールオブジェクトといったインターフェースオブジェクトと，対応するインターフェース要素とを結びつけるものです。

これは，クロークで番号札を渡して自分の預けたコートを受け取るのとよく似ています。番号札のようにハンドルを渡すと目的のウィンドウが得られます。

ウィンドウオブジェクトを使った処理では，ウィンドウハンドルを自分で直接操作する必要はほとんどありません。ただし，Windows 関数を直接呼び出す場合には必要です。たとえば，メッセージボックスを表示させるときは Windows の API 関数 ***MessageBox*** を呼び出します。***MessageBox*** は，メッセージボックスの親ウィンドウへのハンドルを必要とします。以下に示す ***MessegeBox*** の呼び出しは，派生ウィンドウクラスのメンバ関数のように行ないます。Windows の API 関数については **ObjectWindows for C＋＋リファレンスガイド**の第4章『**Windows 関数**』を参照してください。

```
MessageBox(HWindow, "Do you want to save?", "Drawing has changed",
           MB_YESNO | MB_ICONQUESTION);
```

親子関係

ほとんどのウィンドウは，それ自体で独立して存在することはありません。ウィンドウ同士はリンクされ，相互に協調して動作を行ないます。たとえば，あるアプリケーションを終了するとき，このアプリケーションにかかわっていたすべてのウィンドウを破棄しなければなりません。Windows では一般に，親子関係として各ウィンドウをリンクさせています。親ウィンドウはその中にあるすべての子ウィンドウを管理します。ObjectWindows では，各ウィンドウオブジェクトがその親ウィンドウと子ウィンドウを追跡管理できるようにデータメンバを提供しています。

親ウィンドウの関係については76ページ以降のステップ8で説明します。

データメンバ ***Parent*** は，ウィンドウの親ウィンドウオブジェクトを保持します。これは派生クラスに対する基本クラスのような存在ではなく，オーナーウィンドウに近いものです。

3番目のウィンドウデータメンバは**子リスト**です。このデータメンバはウィンドウに**子ウィンドウ**がある場合，この子ウィンドウを指すポインタのリストを保持します。ステップ8で，プログラムに子ウィンドウを追加することにします。

メインウィンドウオブジェクトの作成

ウィンドウオブジェクトによって何が保持されるかがわかったところで，新しいウィンドウクラスを定義することにします。このウィンドウクラスは ***TWindow*** クラスから派生させ，STEP アプリケーションのメインウィンドウとして機能させます。まず，クラス定義を更新して新しいクラス *TMyWindow* を指定します。

STEP 2.CPP の一部です。

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
    TWindow(AParent, ATitle) {};
};
```

新しく派生させたクラスのコンストラクタは新しい動作をなにも定義せず，基本クラス（***TWindow***）のコンストラクタを呼び出すだけです。

次に，*TMyApplication*::*InitMainWindow* を更新し，メインウィンドウとして ***TWindow*** でなく *TMyWindow* を構築します。

STEP 2.CPP の一部です。

```
void TMyApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, Name);
}
```

新しいクラスを宣言し，これを ***InitMainWindow*** でインスタンス化するだけで，*TMyApp* のメインウィンドウ用として新しいクラスが定義されます。アプリケーションオブジェクトは，ウィンドウオブジェクトのメンバ関数を呼び出して，ウィンドウインターフェース要素を作成させ，これを画面に表示させます。プログラマがこの処理を手配する必要はありません。

TMyWindow は，***TWindow*** と ***TWindowsObject*** から継承した動作以外の新しい動作は定義していません。つまり，STEP アプリケーションはまだそれほど実用的ではありません。次の節でいくつかの動作を追加することにしましょう。

メッセージへの応答

ウィンドウオブジェクトを実用的なものにするための一番手っ取り早い方法は，Windows メッセージへの応答方法を教えこむことです。たとえば，STEP アプリケーションのメインウィンドウでユーザがマウスの左ボタンをクリックすると，そのウィンドウオブジェクトは Windows から ***WM_LBUTTONDOWN*** メッセージを受け取ります。これによってウィンドウオブジェクトは，ユーザがウィンドウでマウスをクリックしたことがわかります。さらに，ユーザがクリックした位置の座標も渡されます。これについては，チュートリアルのステップ 6 で説明します。同様に，ユーザがマウスの右ボタンをクリックすると，メインウィンドウオブジェクトは Windows から ***WM_RBUTTONDOWN*** メッセージを受け取ります。

次のステップでは，*TMyWindow* のインスタンスであるメインウィンドウに対し，メッセージに応答して適切な処理を行なう方法を教えこむことにします。Windows メッセージを受け取って応答するには，送られてくるメッセージのうちの応答するメッセージの種別ごとにウィンドウクラスのメンバ関数を定義しておかなければなりません。これを**メッセージ応答メンバ関数**と呼びます。応答する各 Windows メッセージごとにメッセージ応答メンバ関数を定義します（メッセージ応答メンバ関数が定義されていないメッセージに対し，ObjectWindows はデフォルト処理を行ないます)。メンバ関数を応答メンバ関数として識別させるには，その宣言にディスパッチインデックスを付加します（19 ページを参照)。この場合，関数名は意味がなくなり，ディスパッチインデックスが重要になります。メッセージ応答関数名は，Windows メッセージ定数から下線を除去し，一部を大文字に変えた形に統一してあります。

STEP 2.CPP の一部です。

```
class TMyWindow : public TWindow {
public:
:
virtual void WMLButtonDown(RTMessage Msg) =
  [WM_FIRST + WM_LBUTTONDOWN];
:
};
```

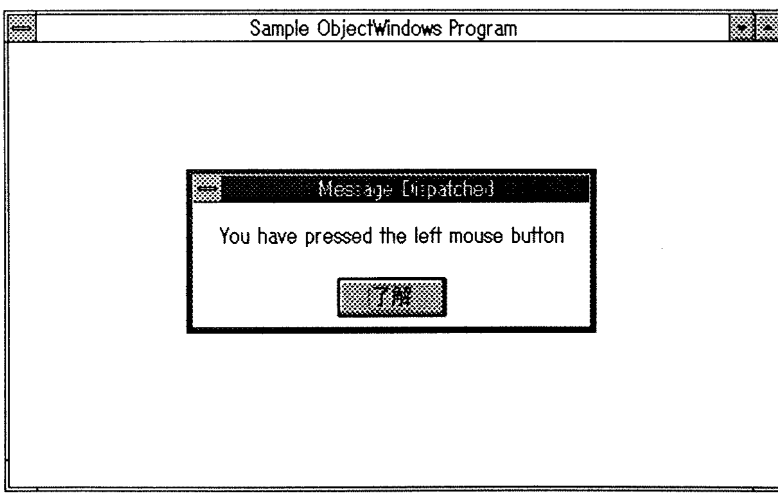
Msg は ***TMessage*** 構造体への参照（ObjectWindows の型 ***RTMessage*** で短略）であり，発生したイベントに関する情報を保持します。この構造体についてはステップ 3 で詳しく説明します。すべてのメッセージ応答関数は ***RTMessage*** を渡されます。リターンタイプは void です。

ここでは，マウスボタンの押下を知らせるメッセージボックスを表示する応答メンバ関数を定義するだけにしておきます。もっと実用的な応答は後で追加することにします。マウスの左ボタンに対する応答メンバ関数の定義を以下に示します。

STEP 2.CPP の一部です。

```
void TMyWindow::WMLButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}
```

図 2.3
ユーザイベントに応答する STEP アプリケーション



このステップのソースコードは，この章の最後にあります。

アプリケーションの終了

ここで作成したプログラムは，メインウィンドウのコントロールメニューボックス（ウィンドウの左上の小さな四角のボックス）をユーザがダブルクリックするとクローズします。ウィンドウとアプリケーションもただちにクローズします。このような動作の仕方は単純なプログラムでは問題ありませんが，複雑なプログラムの場合は不都合な点も出てきます。

たとえば，作業結果をセーブしないでアプリケーションを終了する恐れがあります。常識的なアプリケーションであれば，終了前に作業結果をセーブしていなければ，セーブするかどうかを問い合わせてきます。このような動作はObjectWindowsアプリケーションにも簡単に追加できます。ここでSTEPアプリケーションにユーザの終了要求をダブルチェックさせる機能を追加してみましょう。

ユーザがObjectWindowsアプリケーションをクローズしようとすると，アプリケーションクラスの仮想メンバ関数 ***CanClose*** が呼び出されます。***CanClose*** は論理関数であり，アプリケーションをクローズしてよい（真）かどうかを問い合わせるものです。***TApplication*** から継承したメンバ関数 ***CanClose*** は，デフォルトでメインウィンドウオブジェクトのメンバ関数 ***CanClose*** を呼び出します。たいていの場合，クローズしてよいかどうかを決めるのはメインウィンドウオブジェクトだからです。

メインウィンドウクラス *TMyWindow* は，メンバ関数 ***CanClose*** を ***TWindowsObject*** から継承します。この ***TWindowsObject*** は子ウィンドウがあれば，各子ウィンドウごとにメンバ関数 ***CanClose*** を呼び出します。STEPアプリケーションのように子ウィンドウがない場合，***CanClose*** は単に真を返します。アプリケーションのクローズ時の動作を変更したければ，次のようにメインウィンドウクラスのメンバ関数 ***CanClose*** を定義し直してください。

STEP 2.CPP の一部です。

```
BOOL TMyWindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?",
    "Drawing has changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

これで，ユーザがSTEPアプリケーションをクローズしようとすると，“Do you want to save?”（セーブしますか？）というメッセージボックスが表示されます。「はい(Y)」ボタンをクリックすると偽が返り，メインウィンドウとアプリケーションはクローズされません。「いいえ(N)」ボタンをクリックすると真が返り，アプリケーションは終了します。**図2.4**は変更後のSTEPアプリケーションです。

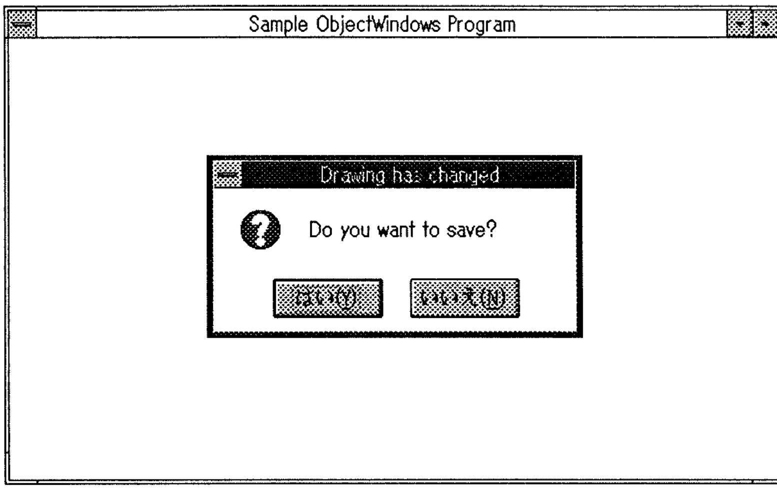


図2.4
クローズ時の動作を改良したSTEPアプリケーション

ここまでのSTEPアプリケーションのソースコードの全体は以下のとおりです。

STEP 2.PASです。

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) : TWindow(AParent,
            ATitle) {};
  virtual BOOL CanClose();
  virtual void WMLButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMRButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_RBUTTONDOWN];
};

BOOL TMyWindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?", "Drawing has
                    changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}

void TMyWindow::WMLButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}

void TMyWindow::WMRButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the right mouse button",
             "Message Dispatched", MB_OK);
}

void TMyApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
   TMyApp MyApp("Sample ObjectWindows Program", hInstance,
                hPrevInstance, lpCmdLine, nCmdShow);
   MyApp.Run();
   return MyApp.Status;
}
```

第3章

ウィンドウへの書き込み

この章では，まだプログラムのシェルにすぎない STEP アプリケーションを実用的な対話型のグラフィックスアプリケーションに変身させましょう。まず，STEP アプリケーションのメインウィンドウにテキストを書き込むことから始めます。次に，STEP アプリケーションを小さなグラフィックスアプリケーションに改造して，メインウィンドウに線を描画できるようにします。これができたら，描画プログラムの総仕上げとして，グラフィックスを描画し，線の太さを変更し，最後にグラフィックスをファイルにセーブして後で再ロードできるようにします。

ディスプレイコンテキストとは

ディスプレイコンテキストは，ウィンドウの描画面に相当するものと考えることができます。Windows がウィンドウにテキストやグラフィックスを描画する際に必要となるものです。したがって，ディスプレイコンテキストを指すハンドル，つまり識別子を Windows グラフィックスデバイスインターフェース(GDI)の描画関数に渡す必要があります。

ウィンドウに描画するには，まずディスプレイコンテキストを取得する必要があります。画面に描画する直前に，ウィンドウオブジェクトのメンバ関数で Windows 関数 ***GetDC*** を次のように呼び出してください。

```
TheDC = GetDC(HWindow);
```

GetDC から返されたディスプレイコンテキストのハンドル（Windows で定義済みの ***HDC*** 型）を今度は Windows の GDI 関数に渡します。GDI 呼び出しの例を次にいくつか示します。

```
LineTo(TheDC, 30, 45);
Rectangle(TheDC, 100, 100, 200, 200);
TextOut(TheDC, 20, 20, "Sample text", 11);
```

テキストやグラフィックスの描画が終ったら，ディスプレイコンテキストを**必ず**解放してください。取得したディスプレイコンテキストは描画が終ったら必ず即時解放してください。

```
ReleaseDC(HWindow, TheDC);
```

☞ 取得したディスプレイコンテキストを解放しておかないと，すぐにディスプレイコンテキストが不足します（Windows 環境全体で使えるディスプレイコンテキストが5つに制限されているためです）。その結果アプリケーションが暴走し，たいていコンピュータがハングアップしてしまいます。描画を3回か4回行なった後でアプリケーションが暴走した場合は，ディスプレイコンテキストが解放されているかどうかを調べてください。

ディスプレイコンテキストには，いくつかの重要な役割があります。まず，ウィンドウ外にテキストやグラフィックスが描画されないようにします。すなわち，GDI 関数を呼び出して表示させたテキストやグラフィックスは，ディスプレイコンテキストを取得したウィンドウの境界に合わせて**クリップ**されます。

ディスプレイコンテキストは，GDI 関数を呼び出したときに使用する描画ツール群（ペン，ブラシ，フォント，パレット）も管理します。前の例では，ディスプレイコンテキストで選択されたペンは，***LineTo*** が呼び出されると線の描画に使われます。描画される線には，選択されたペンの色や幅などの特性が現われます。同じように，ディスプレイコンテキストで選択されたブラシは，***Rectangle*** が呼び出されると画面上の指定矩形領域の描画に使われます。また，選択されたフォントは，***TextOut*** が呼び出されると“Sample text”の表示に使われます。

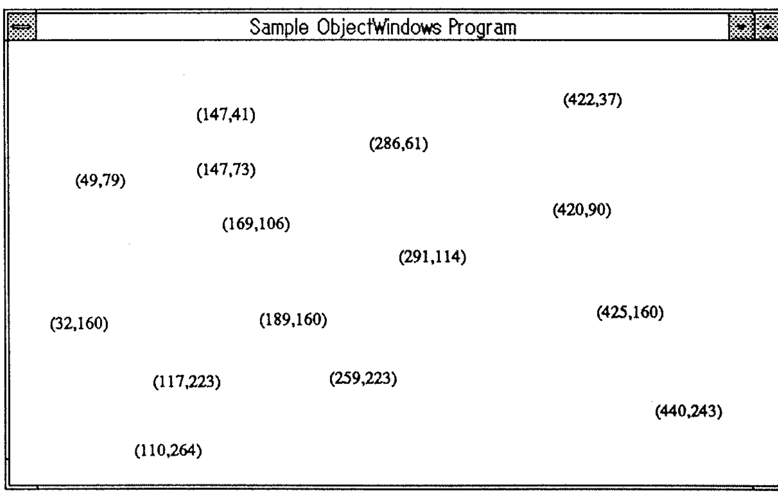
ディスプレイコンテキストに別のツールを登録すると，表示の仕方を変えることができます。たとえば，特定パターンのブラシ，色の異なるペン，スタイルの異なるフォントなどを使うことができます。別のペンを選択する方法はステップ5で述べます。

ステップ 3：ウィンドウにテキストを描画する

STEP アプリケーションのメインウィンドウにテキストを描画する場合でも，「まずディスプレイコンテキストを取得し，次に表示操作を行ない，最後にこれを解放する」という，前の節で説明したディスプレイコンテキストの描画規則に従ってください。ステップ 2 で定義したマウスの左ボタンの機能を変えて，クリックしてテキストを描画できるようにします。今回は，例として，メッセージボックスを表示するかわりに，ウィンドウでクリックした位置の座標を示すテキストを描画することにします。

この例は，Windows の座標系を理解するのにも役立ちます。たとえば，ウィンドウの描画面（クライアント領域）の左上隅を基点にして，右に 20 ピクセル，下に 30 ピクセルの位置でマウスをクリックすると，(20，30) と描画されます。このテキストをクリック位置に描画することもできます。

図 3.1
ユーザがクリックした位置にテキストを描画する STEP アプリケーション



マウスの左ボタンをクリックすると ***WM_LBUTTONDOWN*** メッセージが生成されます。このメッセージは，メッセージ応答メンバ関数 ***WMLButtonDown*** で受け取ります。

メッセージ構造体

ステップ2で説明したように，メッセージ応答メンバ関数に渡される ***TMessage*** 構造体参照（***RTMessage***）である引数 *Msg* には，発生したイベントに関するデータが入っています。このデータは構造体の ***LONG*** 型引数 *lParam* と ***WORD*** 型引数 *wParam* で渡されます。

Msg.LParam は多くの場合，上位ワードと下位ワードが独立して使われ，2つの情報が保持されます。*LParam* は，***WORD*** 型の *Hi* メンバと *Lo* メンバを宣言する ***LP*** 構造体の共用体のメンバとして宣言されます。したがって，*Msg.LP.Hi* と *Msg.LP.Lo* を使えば，*Msg.LParam* の上位ワードと下位ワードを参照できます。同様に，*WParam* は ***WP*** 構造体の共用体のメンバとして宣言されます。ただし，Windows は ***WP*** 構造体の *Hi* メンバと *Lo* メンバをめったに使いません。

WMLButtonDown の場合，*Msg.LP.Lo* はクリック位置のX座標を保持し，*Msg.LP.Hi* はY座標を保持します。クリック位置の座標を描画するように ***WMLButtonDown*** を書き直すには，*Msg.LP.Lo* と *Msg.LP.Hi* をテキストに変換して，カンマを加えます。この例では，**sprintf** を使って描画するテキスト文字列を構築しています。

文字列が完成したら，これをクリックした位置に描画します。Windows の ***TextOut*** 関数を呼び出し，ディスプレイコンテキスト，*Msg.LP.Lo* と *Msg.LP.Hi* の位置座標，文字列 *S*，文字列長 **strlen**（*S*）を渡してください。なお，ディスプレイコンテキストを描画前に取得し，描画後すぐに解放することを忘れないでください。

このコードはSTEP3.CPPの一部です。

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  HDC DC;
  char S[10];

  sprintf(S, "(%d,%d)", Msg.LP.Lo, Msg.LP.Hi);
  DC = GetDC(HWindow);
  TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, S, strlen(S));
  ReleaseDC(HWindow, DC);
}
```

画面のクリア

テキスト描画アプリケーションに追加できる機能として，画面のクリアがあります。もっとも，ウィンドウの大きさを変えたり，ウィンドウの非表示や表示を行なえば，描画されているテキストは消えます。しかし，メニュー選択やマウスのクリックなどの操作で強制的に画面をクリアしなければならない場合もあります。

ここでは，マウスの右ボタンのクリックでウィンドウをクリアすることにします。この場合，メンバ関数 ***WMRButtonDown*** を定義し直して Windows 関数 ***InvalidateRect*** を呼び出し，その引数でウィンドウ全体の再描画を指示します。現時点ではプログラム全体の動きが明確でなく，ウィンドウの再描画方法が不明なので，クライアント領域をクリアするだけにとどめます。

このコードはSTEP3.CPPの一部です。

```
void TMyWindow::WMRButtonDown(RTMessage)
{
  InvalidateRect(HWindow, NULL, TRUE);
}
```

このソースコードは STEP3.CPP ファイルにあります。

ステップ 4：ウィンドウに線を描画する

ディスプレイコンテキストの取得，描画，ディスプレイコンテキストの解放という典型的な描画方法を習得したので，この方法をさらに高度な対話型のグラフィックスアプリケーションに応用することにしましょう。この節では，ユーザがメインウィンドウで描画できる簡単なペイントプログラムを次の手順で作成することにします。

1. マウスの左ボタンのクリックとドラッグに応答し，軌跡に沿ってドットを連結し，線を描画する。
2. マウスの右ボタンのクリックに応答して入力ダイアログを表示し，ユーザが線の太さを変更できるようにする。
3. 点座標を格納しておき，**ペイント**メッセージが来たらこれを再描画してウィンドウの内容を再描画する。

以上の作業を行なうために，まず Windows のドラッグモデルを理解し，次に簡単なグラフィックス描画プログラムを実現することにしましょう。

ドラッグモデル

最初に典型的な Windows のマウスイベント処理を復習しておきましょう。第2章で DDVT サポートを使って説明したように，マウスの左ボタンをクリックすると，***WM_LBUTTONDOWN*** メッセージが生成され，メンバ関数 ***WMLButtonDown*** を呼び出します。このチュートリアルの初めのほうでは，マウスの左ボタンをクリックすると，メッセージボックスが表示されたり画面にテキストが描画されたりしました。また，マウスの右ボタンをクリックすると，***WM_RBUTTONDOWN*** メッセージが生成され，メンバ関数 ***WMRButtonDown*** を呼び出し，画面を消去しました。しかし，この2つの処理はマウスボタンの初回のクリックしか扱っていません。多くの対話型 Windows プログラムでは，マウスをクリック，ドラッグして画面に線や矩形を描いたり，グラフィックスを移動させなければなりません。これから作成するグラフィックス描画プログラムは，ドラッグイベントを取得し，イベントに応じて線を描画できるようにしてみましょう。

このような処理を行なうには，さらにいくつかのメッセージに応答する必要があります。ユーザがマウスをウィンドウ内の別の位置にドラッグすると，***WM_MOUSEMOVE*** を受信し，マウスの左ボタンを放すと ***WM_LBUTTONUP*** を受信します。通常，ウィンドウは ***WM_LBUTTONDOWN*** メッセージを1つ受け取り，その後に一連の ***WM_MOUSEMOVE*** メッセージを受け取り，最後に ***WM_LBUTTONUP*** メッセージを1つ受け取ります。Windows のグラフィックスプログラムは，***WM_LBUTTONDOWN*** を受け取ると描画プロセス（ディスプレイコンテキストの取得を始めとする一連の初期化処理）を開始します。***WM_MOUSEMOVE*** を受け取るとグラフィックスの描画と移動を行ない，また ***WM_LBUTTONUP*** を受け取ると描画プロセスを終了します（ディスプレイコンテキストの解放）。

一般的なマウスイベントメッセージを次の表にまとめました。

表 3.1
一般的なマウスイベントメッセージ

メッセージ	イベント
WM_LBUTTONDOWN	マウスの左ボタンのクリック
WM_RBUTTONDOWN	マウスの右ボタンのクリック
WM_MOUSEMOVE	マウスのドラッグ
WM_LBUTTONUP	マウスの左ボタンの解放
WM_RBUTTONUP	マウスの右ボタンの解放
WM_LBUTTONDBLCLK	マウスの左ボタンのダブルクリック
WM_RBUTTONDBLCLK	マウスの右ボタンのダブルクリック

ドラッグメッセージへの応答

線を描画するプログラムを作成する際の，ドラッグメッセージに対する応答内容を次の表にまとめました。

表 3.2
ステップ 4 で使用するメッセージ

メッセージ	応答内容
WM_LBUTTONDOWN	画面をクリアする。次にディスプレイコンテキストを取得して *DragDC* に格納する。クリック位置に描画用のペンを配置する。
WM_MOUSEMOVE	マウスボタンが押下のままであれば，前の位置から現在の位置まで線を描く。
WM_LBUTTONUP	*DragDC* を解放する。

すでに説明したように，***WM_LBUTTONDOWN*** の後には必ず ***WM_LBUTTONUP*** が続きます。この 2 つのメッセージ間に ***WM_MOUSEMOVE*** メッセージが入る場合もあります。ウィンドウが ***WM_LBUTTONDOWN*** メッセージを受け取ったときにディスプレイコンテキストを取得し，***WM_LBUTTONUP*** を受け取ったときにディスプレイコンテキストを解放してください。このプログラムでは，ユーザがマウスの左ボタンをクリックして離すまでの間に行なわれるすべての描画に対して，ディスプレイコンテキストを 1 つだけ使います。

ディスプレイコンテキストを取得したら解放するという規則は，グラフィックスプログラムを正常に機能させるために不可欠です。さらに安全策を講じることもできます。すなわち，*ButtonDown* という新しい論理型データメンバをメインウィンドウクラス *TMyWindow* に定義しておきます。*ButtonDown* は，***WMLButtonDown*** により真に設定され，***WMLButtonUp*** によって偽に設定されます。こうしておけば，ディスプレイコンテキストの取得または解放の前に *ButtonDown* の値を調べることで，マウスボタンの状態を知ることができます。

3種類のマウスドラッグ用メンバ関数を以下に示します。

このコードはSTEP 4 .CPPの一部です。

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
  }
}

void TMyWindow::WMMouseMove(RTMessage Msg)
{
  if ( ButtonDown )
    LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
}

void TMyWindow::WMLButtonUp(RTMessage)
{
  if ( ButtonDown )
  {
    ButtonDown = FALSE;
    ReleaseCapture();
    ReleaseDC(HWindow, DragDC);
  }
}
```

MoveTo と ***LineTo*** は，Windows APIのグラフィックス関数です。前者は現在の描画位置を移動し，後者は現在の位置まで直線を描画します。この2つの関数には，*TMyWindow* がデータメンバ *DragDC* に格納しているディスプレイコンテキストへのハンドルが必要です。ウィンドウに直接描画するわけではなく，ディスプレイコンテキストに描画するからです。

マウスが *TMyWindow* の外側までドラッグされ，マウス位置が隣接するウィンドウに入った場合でも，***WM_MOUSEMOVE*** メッセージは *TMyWindow*

に送られます。これは，*TMyWindow* が **SetCapture** を呼び出してマウス入力を取得するためです。なお，マウス入力を解放するときは ***ReleaseCaputre*** を呼び出します。

TMyWindow のオブジェクト定義は ***WMMouseMove*** と ***WMLButtonUp*** のメンバ関数ヘッダで必ず更新してください。

STEP 4.CPP のコード内容

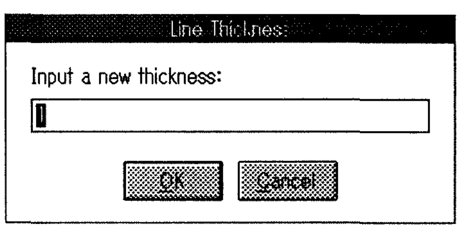
```
virtual void WMLButtonDown(RTMessage Msg)
  = [WM_FIRST + WM_LBUTTONDOWN];
virtual void WMLButtonUp(RTMessage Msg)
  = [WM_FIRST + WM_LBUTTONUP];
```

ステップ 5：線の太さを変更する

これまでのプログラムでは，細い線しか描けませんでした。一般の描画プログラムでは線の太さを変えられるので，*MyProgram* にもこの機能を加えることにしましょう。この処理は，実際に線の太さを変えるのではなく，線の描画に使うペンの太さを変えるものです。ペン，ブラシ，フォント，パレットは，ディスプレイコンテキストによって実現される描画ツールです。このステップでは，STEP アプリケーションに線の太さを変える機能を追加しながら，新しいツールをディスプレイコンテキストに設定する方法を学んでいきます。

さらに，入力ダイアログ（***TInputDialog*** クラス）を使って，ユーザが線の太さを簡単に変更できるようにします。**図 3.2** に使用する入力ダイアログを示します。

図 3.2
線の太さの変更



新しいペンの選択

描画する線の太さを変更する前に，Windows グラフィックス，特にディスプレイコンテキストについてもう少し詳しく理解しておく必要があります。

グラフィックスやテキストをウィンドウに作成する描画ツールには，ペン，ブラシ，フォントがあります。これらの描画要素の記述内容は，インターフェース要素と同様に Windows メモリに格納されています。描画要素もインターフェース要素もハンドルで識別されます。ただし，描画要素は ObjectWindows オブジェクトではないので，プログラムで使用する描画ツールの作成と削除はプログラムで行ないます。

☞ ツールの削除を忘れると，Windows メモリのかなりの量が占有されたままになります。

描画ツールとディスプレイコンテキストは画家の絵筆とキャンバスにたとえることができます。画家はすべてのツール（絵筆）とディスプレイコンテキスト（キャンバス）を用意すると，次に使用する描画ツールを選びます。同様に，Windows プログラムもディスプレイコンテキストで使う描画ツールを選択する必要があります。描画ツールを選択しなければ，テキストや線をウィンドウに描くことはできないのです。ディスプレイコンテキストはどれでも，デフォルトのツール一式が揃っています。黒色の細ペン，白ベタのブラシ，およびシステムフォントがデフォルトツールです。このステップでは，別のもっと太いペンを使ってウィンドウに描画してみましょう。

まず，入力ダイアログ用の機能を STEP アプリケーションに追加することにします。入力ダイアログ用の機能を定義したヘッダファイルは INPUTDIA.H です。これを.CPP ソースファイルに追加してください。入力ダイアログでは，ダイアログリソースが.RC ファイルに定義されていることを前提としています。入力ダイアログリソースは INPUTDIA.DLG に定義されています。このファイルを.RC ファイルにインクルードしてください。STEPS.RC には STEP アプリケーションに必要なすべてのリソース定義が入っています。.RC ファイルには次のファイルがインクルードされていなければなりません。

```
#include "windows.h"
#include "owlrc.h"
rcinclude inputdia.dlg
```

次に，*TMyWindow* にデータメンバを追加して，グラフィックスの描画に使うペンツールのハンドルを格納します。このプログラムでは，同時に描画，表示する線の太さを１種類に限ることにします。この太さのペンを，*TMyWindow* の新しいデータメンバ *ThePen* に格納します。さらに，新しいペンツールを作成し古いペンツールを削除するメンバ関数 *SetPenSize* を記述します。完成した *TMyWindow* クラスの宣言は次のようになります。

このコードは STEP 5 .CPP の一部です。

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  HDC DragDC;
  BOOL ButtonDown;
  HPEN ThePen;
  int PenSize;
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
  ~TMyWindow();
  virtual BOOL CanClose();
  void SetPenSize(int NewSize);
  virtual void WMLButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMLButtonUp(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONUP];
  virtual void WMMouseMove(RTMessage Msg)
    = [WM_FIRST + WM_MOUSEMOVE];
  virtual void WMRButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_RBUTTONDOWN];
};
```

新しいデータメンバを初期化するため，コンストラクタを変更してペンを設定させ，デストラクタを定義してペンを破棄させてください。

このコードは STEP 5 .CPP の一部です。

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
}

TMyWindow::~TMyWindow()
{
  DeleteObject(ThePen);
}
```

次に，メンバ関数 ***WMLButtonDown*** を変更して，現在のペン（*ThePen*）を新たに取得したディスプレイコンテキストに採用します。

このコードはSTEP 5 .CPPの一部です。

MoveToとMessage-Box同様，SelectObjectもWindows API関数です。

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
  }
}
```

ペンの変更

このメンバ関数は，作成済みのペンをディスプレイコンテキストに採用するものです。ペンを新たに作成する場合は，次のメンバ関数 *SetPenSize* を記述する必要があります。

このコードはSTEP 5 .CPPの一部です。

```
void TMyWindow::SetPenSize(int NewSize)
{
  DeleteObject(ThePen);
  ThePen = CreatePen(PS_SOLID, NewSize, 0);
  PenSize = NewSize;
}
```

Windows関数 ***CreatePen*** を呼び出すというのは，線の太さを指定してWindowsペンツールを作成する方法の1つです。ペンを指すハンドルは ***ThePen*** に格納しておきます。以前に使用したペンは必ず削除してください。削除しないと，Windowsメモリを徐々に消費し尽してしまいます。

入力ダイアログの実行

線の太さを変更するとき，マウスの右ボタンをクリックして変更用オプションを表示できれば便利です。そこで，入力ダイアログが現われるようにメンバ関数 ***WMRButtonDown*** を定義し直すことにします。このダイアログは

ObjectWindowsのストックダイアログ（あらかじめ用意されているダイアログ型）の1つです。入力ダイアログボックスはユーザが入力した1行のテキストを取得するだけの簡単なダイアログボックスです。

入力ダイアログオブジェクトを構築したら，***TModule***::***ExecDialog***. ***ExecDialog***を呼び出してモードつきダイアログとして実行させることができます。ダイアログオブジェクトに用いる***ExecDialog***は，ウィンドウオブジェクトとモードなしダイアログボックスとして使うダイアログオブジェクトに用いる***MakeWindow***に似ています。両方とも，インターフェースオブジェクトを使える場合に限ってインターフェース要素を作成します。ただし，***ExecDialog***は，ユーザがOKまたはCancelをクリックしてダイアログをクローズしなければ終了しません。ユーザがOKをクリックすると*InputText*にはユーザの入力した内容が入ります。太さ番号を要求しているので，返されたテキストを数値に変換し，*SetPenSize*に渡さなければなりません。ユーザが新たに線の太さを選択するたびに，古いペンを削除して新しいペンを作成します。

この時点までのプログラムのコードはSTEP5.CPPにあります。

```
void TMyWindow::WMRButtonDown(RTMessage)
{
  char InputText[6];
  int NewPenSize;

  sprintf(InputText, "%d", PenSize);
  if ( GetApplication()->ExecDialog(new TInputDialog(this, "Line
                                    Thickness", "Input a new
                                    thickness:", InputText, sizeof
                                    InputText)) == IDOK )
  {
      NewPenSize = atoi(InputText);
      if ( NewPenSize < 0 )
        NewPenSize = 1;
     SetPenSize(newPenSize);
  }
}
```

ステップ6：ウィンドウを再描画する

TextOut や ***LineTo*** などの Windows 関数を使ってウィンドウに描画したグラフィックスやテキストは，ウィンドウのサイズを変更したり，ウィンドウを覆っている別のウィンドウを取り去ると消えてしまいます。一体，どこに行ってしまったのでしょうか。

この問題については,「グラフィックスやテキストが最初はどこにあったのか」ということを考えてみるといいでしょう。実は，テキストや線は，いかなる型の変数にも格納されていなかったのです。したがって，Windows 関数を呼び出していったんグラフィックスデータを Windows に渡してしまうと，後で必要になってもこれを呼び戻して再描画することはできないのです。ウィンドウにグラフィックスを再表示させるには，グラフィックスを別の型の構造体に格納しておく必要があります。この場合はオブジェクトにしておくのがいいでしょう。オブジェクトは，単純なグラフィックスでも複雑なグラフィックスでも格納でき，またメインウィンドウのデータメンバとして管理しやすいためです。

ペイントモデル

ウィンドウの表示を更新，つまりペイントする必要があるときは，Windows からアプリケーションに ***WM_PAINT*** メッセージが送られます。たとえば，ユーザがウィンドウのサイズを変更したり，覆っているウィンドウを取り除いたりすると，***WM_PAINT*** メッセージが送られます。ObjectWindows は，この ***WM_PAINT*** メッセージを受け取り，ウィンドウのメンバ関数 ***WMPaint*** を呼び出します。

TWindowsObject は，Windows のセットアップとクリーンアップの前後に仮想関数 ***Paint*** を呼び出す ***WMPaint*** を定義します。***TWindow*** の派生クラスでは ***Paint*** を定義し直してウィンドウの表示を描画させてください。

メンバ関数 ***Paint*** でグラフィックスを描画するのと，マウス操作でグラフィックスを描画するのとでは大きな違いが1つあります。***Paint*** で使われるディスプレイコンテキストは，引数 *PaintDC* で渡されるので，コンテキストの取得と解放をプログラムで行なう必要はありません。ただし，描画ツールを *PaintDC* で選択し直さなくてはなりません。

ウィンドウの内容をペイントする場合，*DragDC* に基づいた最初の内容を再描画せず *PaintDC* を使ってください。描画内容は最初にユーザが描画したのと同じように見えます。すなわちコンサートの録音を再生するのに似ています。実際に描画しているのか，ObjectWindows が行なっているのかは区別できません。メンバ関数 ***Paint*** が必要とするグラフィックステータをまず格納しておいてください。

オブジェクトとしてのグラフィックスの格納

ステップ4では，もとの線が ***WM_MOUSEMOVE*** メッセージに対する処理として描画されることを説明しました。***WM_MOUSEMOVE*** メッセージは，マウスの現在位置をウィンドウディスプレイ上の点のX座標とY座標として示します。各 ***WM_MOUSEMOVE*** メッセージで送られてきた点を連結して線が描画されました。メッセージで受け取った各点が，線を再描画するのに必要なグラフィックステータです。

各点を表わすのに，***Object*** の派生クラスである *TPoint* オブジェクトを使うことにします。ここでオブジェクトのかわりに簡単な構造体を使ってもいいのですが，*TPont* オブジェクトをコンテナクラスライブラリのいずれかのコンテナに格納することにします。コンテナクラスライブラリのコンテナはどれも ***Object*** から派生したオブジェクトを要求します。また以降のステップで，*TPoint* にメンバ関数を追加することになります。

TPoint オブジェクトを *Points* という *TMyWindow* のデータメンバに格納することにします。*Points* はコンテナクラス **Array** から派生した *TPointArray* 型のインスタンスです。**Array** ではなく *TPointArray* を使う理由は，**Array** にない新しい機能を *TPointArray* に追加するためです。**Array** オブジェクト（および **Array** からの派生クラス）は動的に成長する方法を心得ているので，不特定数の点を保持するのに大変適しています。**Array** オブジェクトと他のコンテナクラスの説明はコンパイラのマニュアルにあります。以下は *TPoint* の定義の一部です。

```
class TPoint: public Object {
public:
  int X, Y;
  TPoint(int AX, int AY) {X = AX, Y = AY;}
⋮
};
```

次は *TPointArray* 宣言の一部です。

このコードはSTEP 6 .CPP の一部です。

```
_CLASSDEF(TPointArray)

class TPointArray : public Array
{
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0) :
              Array(upper, lower, aDelta) {};
  void DeleteAll();
⋮
};
```

データメンバ *Points* を追加して更新した *TMyWindow* の宣言を次に見てみましょう。

このコードはSTEP 6 .CPP の一部です。

```
_CLASSDEF(TPointArray)

class TPointArray : public Array
{
⋮
ointArray Points;
⋮
};
```

Points は必ず *TMyWindow* のコンストラクタで構築し，*TMyWindow* のデストラクタで廃棄してください。

このコードはSTEP 6.CPPの一部です。

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}

TMyWindow::~TMyWindow()
{
  delete Points;
  DeleteObject(ThePen);
}
```

Points を設定するには，***WMLButtonDown*** と ***WMMouseMove*** を次のように変更しなければなりません。

このコードはSTEP 6.CPPの一部です。

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}

void TMyWindow::WMMouseMove(RTMessage Msg)
{
  if ( ButtonDown )
  {
    LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}
```

ArrayIteratorの使用法についてはコンテナクラスライブラリの資料を参照してください。

WMLButtonUp を変更する必要はありません。*TMyWindow*::*WMLButtonDown* では，新しい線を描画する前に *TPointArray*::*DeleteAll* メンバ関数を呼び出して *Points* **Array** から *TPoint* を消去させます。

このコードはSTEP 6 .CPPの一部です。

```
void TPointArray::DeleteAll()
{
  RArrayIterator PointIterator = (RArrayIterator)initIterator();

  while ( int(PointIterator) != 0 )
  {
    RObject AnObject = PointIterator++;
    if ( AnObject != NOOBJECT )
      detach(AnObject, 1);  // 分離して削除
  }
  delete &PointIterator;
}
```

格納してあるグラフィックスの再描画

Points が設定されたので，*TMyWindow* をペイントするコードを記述し，点と点の間に線を再描画します。ここでは *TMyWindow* のメンバ関数 ***Paint*** を記述して，メンバ関数 ***WMLButtonDown*** と ***WMMouseMove*** で行なう動作をデータメンバ *Points* を使って繰り返させます。メンバ関数 ***Paint*** を以下のように記述します。

このコードはSTEP 6 .CPPの一部です。

```
void TMyWindow::Paint(HDC DC, PAINTSTRUCT&)
{
  RArrayIterator PointIterator =
                 (RArrayIterator)(Points->initIterator());
  BOOL First = TRUE;

  SelectObject(DC, ThePen);
  while ( int(PointIterator) != 0 )
  {
    RObject AnObject = PointIterator++;
    if ( AnObject != NOOBJECT )
    {
      if ( First )
      {
        MoveTo(DC, ((PTPoint)(&AnObject))->X,
                   ((PTPoint)(&AnObject))->Y);
        First = FALSE;
      }
      else
        LineTo(DC, ((PTPoint)(&AnObject))->X,
                   ((PTPoint)(&AnObject))->Y);
    }
  }
delete &PointIterator;
}
```

いままでの内容をまとめます。STEP アプリケーションのメインウィンドウは，*TPointArray* を指すポインタをデータメンバ *Points* に保持します。ユーザが線を描画すると，ウィンドウはマウス位置を示す ***WM_MOUSEMOVE*** メッセージを受信します。この位置データを入れる *TPoint* オブジェクトを構築し，各 *TPoint* を *Points* 配列に追加しておきます。こうしておくと，ウィンドウの再描画が必要なとき，各 *TPoint* を結合して線を再描画できます。

第4章

メニューの追加

Windowsアプリケーションのほとんどのメインウィンドウにはメニューがあり，ユーザはFile | Save，File | Open，Helpなど，各種の選択を行なえます。この章では，STEPアプリケーションに標準メニューを追加してみましょう。

ウィンドウを使った環境では，メニューの選択はマウスのクリックと同じ分類に入り，ともにユーザイベントになります。したがって，メニュー選択に対する応答は，他のユーザイベントに対する応答に似ています。この章では，以下の手順でアプリケーションにメニューを追加することにします。

1. メニューをメニューリソースとして設計する。
2. メニューリソースをメインウィンドウオブジェクトにロードする。
3. メニュー選択に対する応答を定義する。
4. アプリケーションをコンパイルし，実行形式ファイルを作成する。
5. メニューリソースを実行形式ファイルにつけ加える。

メニューリソース

アプリケーションでメニューを使っている場合，メニュー項目のテキストおよび最上位項目と下位項目の構造を指定するメニュー仕様が必要です。また，多くのアプリケーションにメニューオプション用のキーボードショートカット**アクセラレータ**が提供されています。この機能のついたアプリケーションでユーザは，たとえばマウスでEdit | Pasteを選択するかわりに***SHIFT-INS***を押して同じ処理を行なわせることができます。

メニューとそのアクセラレータはプログラムのC++ソースコードの一部ではなく，**リソース**という別の仕様に含まれるものです。Windowsはコンパクトで効率的な方法でリソースを格納します。

アプリケーションは，リソースIDを指定してアプリケーション付属のリソースにアクセスします。このIDには，100のような整数か，*MyMenu*のような文字列識別子を使います。アプリケーションは，それぞれのメニュー項目に対応したメニューIDによってメニュー選択を特定します。

メニューリソースを作成するには，リソースエディタか，RC（コマンドラインリソースコンパイラ）を使います。STEPアプリケーションのメニューリソースはソースコード形式でSTEPS.RCファイルに収めてあります。Borland C++ IDEを使用していれば，このSTEPS.RCをSTEPアプリケーションのプロジェクトに追加することができます。IDEは.RCを使ってそれを自動的に翻訳します。以下にSTEPS.RCを示します。

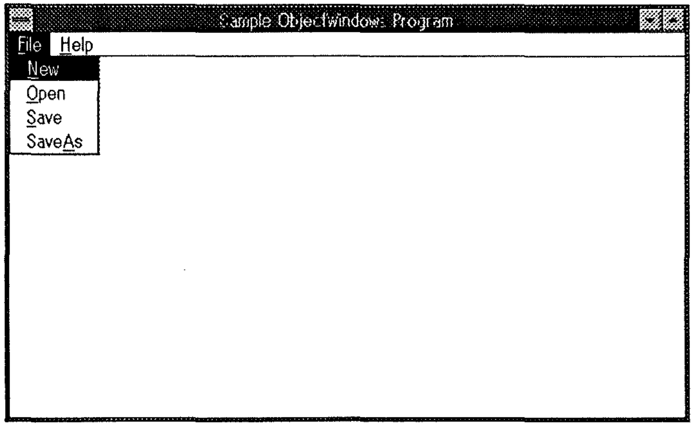
```
#include <windows.h>
#include <owlrc.h>
#include "steps.h"

rcinclude inputdia.dlg
rcinclude filedial.dlg

COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MenuItem  "&New", CM_FILENEW
    MenuItem  "&Open", CM_FILEOPEN
    MenuItem  "&Save", CM_FILESAVE
    MenuItem  "Save&As", CM_FILESAVEAS
  END
  MenuItem  "&Help", CM_HELP
END
```

図4.1は，このメニュー（リソースID "COMMAND"）の外観です。**Help**項目（メニューID ***CM_HELP***）と**File**項目があり，**File**項目はさらに下位項目として**New**，**Open**，**Save**，**SaveAs**（メニューID ***CM_FILENEW***，***CM_FILEOPEN***，***CM_FILESAVE***，***CM_FILESAVEAS)***があります。**File**のように下位項目のある最上位メニューにはメニューIDがありません。したがって，最上位メニューを選択しても何も処理は行なわれず，下位項目が表示されるだけです。**Help**のように下位項目のない最上位メニューにはメニューIDがあり，なんらかの処理が行なわれます。

図 4.1
メニューリソースのついた STEP アプリケーション



メニューの設計は ObjectWindows プログラムの外部で行なわれ，しかもメニュー処理はとても単純なものなので，メニューをオブジェクトにする必要はありません。メニューは，メインウィンドウのタイトルのように，メインウィンドウの属性にしておきます。ステップ 7 で，メニューをメインウィンドウに追加することにします。

ステップ 7：メニューを追加する

すでに説明したように，アプリケーションのメニューは独立したオブジェクトではなく，メインウィンドウが所有しているものです。事実，オブジェクトとはほど遠い存在で，メインウィンドウの属性にすぎません。メニューが格納されるのは *Attr* の *Menu* フィールドであり，このフィールドはウィンドウの**作成属性**を入れるウィンドウオブジェクトのデータメンバです。まず ***AssignMenu*** を呼び出してウィンドウのコンストラクタでメニュー属性を設定してください。

STEPS.RC に格納されているメニューリソースのリソース ID は "COMMAND" です。***AssignMenu*** の呼び出しは以下のように行ないます。

```
AssignMenu("COMMANDS");
```

メニューリソースに整数リソース ID を使う場合は，オーバーロードしたメンバ関数 ***AssignMenu*** を以下のように使えます。

```
AssignMenu (100);
```

TMyWindow のコンストラクタを以下に示します。***TWindow*** のコンストラクタを呼び出して，すべてのウィンドウオブジェクトに必要な初期化を行なっている点に注目してください。

このコードはSTEP7.CPPの一部です。

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                          TWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}
```

これで，メインウィンドウが現われれば，**図4.1** に示したメニューが表示されます。ただし，メニュー選択に応じて実際の処理を行なわせるには，以降のステップに従って，メニューの選択に応答させる必要があります。

メニューメッセージの受け取り

ユーザがウィンドウのメニューから項目を選択するか，またはアクセラレータを使うと，そのウィンドウに Windows の**コマンドメッセージ**が送られます。特定のメニュー項目が選択されたときに送られるコマンドメッセージを処理するには，ObjectWindows の**コマンド応答メンバ関数**を定義しなければなりません。コマンド応答メンバ関数に対し，以下の例のように，特別な関数宣言を使ってください。

```
virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
```

メッセージの範囲とオフセットについては第7章で詳しく説明します。

CM_FIRST は ObjectWindows で定義されている定数であり，該当するメニュー項目の ID を表わします。上の例では，*CM_FILENEW* は OWLRC.H で定義されているので，改めて定義する必要はありません。しかし，Help のように固有のメニュー項目を定義する場合は，**#define** *CM_HELP 201* などの定数を定義しなければならない場合もあります。定義したリソース ID 定数はす

べてヘッダファイルに入れて，.CPP ファイルと，RC ファイルにインクルードしてください。なお，コマンド応答メンバ関数の識別には専用の ***CM_FIRST*** 拡張子が使われます。この拡張子を，ウィンドウのメッセージ応答メンバ関数に使われる ***WM_FIRST*** 拡張子と混同しないでください。

すべての応答メンバ関数に ***TMessage*** 構造体の参照（***RTMessage*** 型）が引数として渡されます。ただし，この構造体はコマンド応答メンバ関数ではあまり使われません。前に述べたように *Msg.WParam* には選択したメニューの ID が入ります。*Msg.LP.Hi* は通常無視します。メニュー項目がアクセラレータで選択されたときのメッセージの場合，*Msg.LP.Hi* には 1 が入ります。*Msg.LP.Lo* には常に 0 が入ります。これですべてのコマンドメッセージの応答メンバ関数を，上で述べたリソース ID 定数を使って宣言できます。なお，*CM_HELP* が定義されているのは STEPS.H であり OWLRC.H ではありません。

このコードは STEP 7 .CPP の一部です。

```
virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
virtual void CMFileOpen(RTMessage Msg) = [CM_FIRST + CM_FILEOPEN];
virtual void CMFileSave(RTMessage Msg) = [CM_FIRST + CM_FILESAVE];
virtual void CMFileSaveAs(RTMessage Msg) = [CM_FIRST +
                                            CM_FILESAVEAS];
virtual void CMHelp(RTMessage Msg) = [CM_FIRST + CM_HELP];
```

メニューメッセージへの応答

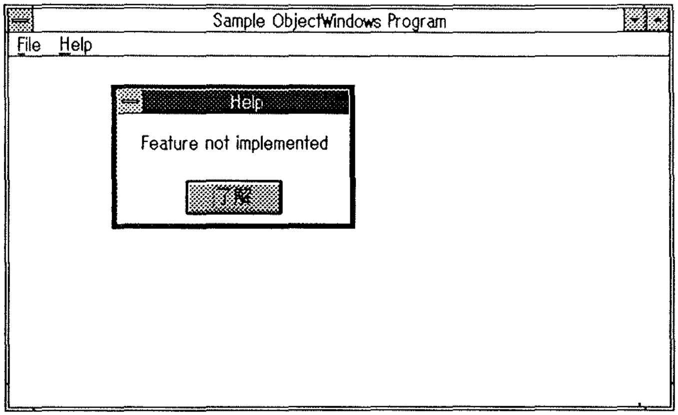
以上で各メニュー項目に対して呼び出されるメンバ関数が用意されました。たとえば，**Help** を選択するとメンバ関数 *CMHelp* が呼び出されます。次のように記述するとメッセージボックスを表示できます。

このコードは STEP 7 .CPP の一部です。

```
void TMyWindow::CMHelp(RTMessage)
{
  MessageBox(HWindow, "Feature not implemented", "Help", MB_OK);
}
```

図 4.2 は Help 項目の選択に対する STEP アプリケーションの応答です。

図 4.2
ヘルプを備えたSTEPアプリケーション



今度は画面をクリアして **File | New** のメニュー選択に応答してみましょう。次のようにメンバ関数 *CMFileNew* を追加してください。

このコードはSTEP 7.CPP の一部です。

```
void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
}
```

このメンバ関数は格納されている点をすべて削除し，画面を強制的にリペイントします。再描画すべき点が残っていないので画面はブランクになります。

CM_OPEN，***CM_SAVE***，***CM_SAVEAS*** の各メッセージに対しては *CMHelp* に似たダミーのメンバ関数を記述しておきます。このダミーのメンバ関数は実際の処理に合わせて後で書き直します。

ここまでの STEP アプリケーションのソースコードは STEP 7.CPP ファイルに収めてあります。

第5章

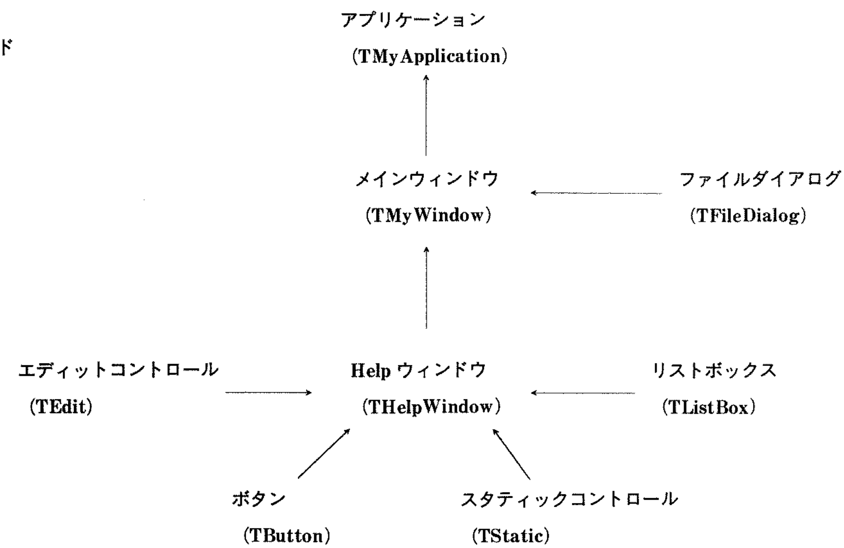
ダイアログの追加

充分な機能を持ったWindowsプログラムであれば，ウィンドウ，コントロール，ダイアログなど，各種のウィンドウオブジェクトを備えることができます。メインウィンドウ以外のウィンドウ，すなわち，子ウィンドウにはそれぞれ親ウィンドウが1つあります。1つの親ウィンドウに多数の子ウィンドウがあることもあります。

このようなウィンドウの親子関係は各アプリケーションで保たれており，メインウィンドウが最終的な親になります。したがって，それぞれのウィンドウが他のウィンドウの親であったり子であったりします。STEPアプリケーションにおけるウィンドウの親子関係を**図5.1**に示します。この関係は，今のところは部分的にしか実現されていません。

子ウィンドウは2種類に分かれます。1つは独立子ウィンドウで，表示形式や位置をそれ自体で制御します。メッセージボックスとポップアップウィンドウはこの種のウィンドウです。もう1つは従属子ウィンドウで，親ウィンドウに重なって表示され，親ウィンドウとともに移動します。メッセージボックスのボタンを始めとするコントロールなどが従属子ウィンドウです。ここでは，ポップアップウィンドウとダイアログボックスという2種類の独立子ウィンドウオブジェクトを作成することにします。

図 5.1
親子関係にあるウィンドウ群



親と子は**所有**関係にあります。たとえば，ステップ 8 では，メインウィンドウがヘルプウィンドウを**所有**しています。Windows における所有関係の考え方は，オブジェクト指向プログラミングでの所有関係，すなわちインスタンスの連結という考え方とほとんど同じです。このチュートリアルでも，ほとんどの場合，子ウィンドウオブジェクトは，その親ウィンドウオブジェクトのデータメンバに格納されます。ここに格納しなければならないというわけではありませんが，プログラムがわかりやすくなり，保守も簡単になるのでお勧めできる設計技法です。

ステップ 8：ポップアップウィンドウとダイアログボックスを追加する

ここでは，Help メニューを選択すると表示されるポップアップウィンドウを STEP アプリケーションに追加してみましょう。このウィンドウは，STEP アプリケーションで使われるヘルプシステムの基礎になります。とりあえず，***TWindow*** のインスタンスとしてポップアップウィンドウを作成しておきます。ステップ 10 でこれに新しいクラスを作成し，動作を定義します。

ここで作成するポップアップウィンドウはメインウィンドウに追加される最初の新ウィンドウです。このウィンドウを作成しながら，ウィンドウオブジェクトとウィンドウ要素をどのように作成して表示するかを学んでいきます。

ポップアップウィンドウの作成

ステップ7では，Helpメニューの選択に対する応答としてメッセージボックスを表示させました。今度は，このメッセージボックスのかわりに空白のポップアップウィンドウオブジェクト ***HelpWindow*** を使います。応答結果はメッセージボックスよりもつまらないものになりますが，ステップ10でこのHelpウィンドウに便利な動作がたくさん追加されます。

とりあえず，次のように *CMHelp* を定義し直してHelpウィンドウが作成されるようにします。

このコードはSTEP8.CPPの一部です。

```
void TMyWindow::CMHelp(RTMessage)
{
  PTWindow HelpWindow;

  HelpWindow = new TWindow(this, "Help System");
  HelpWindow->Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  HelpWindow->Attr.X = 100;
  HelpWindow->Attr.Y = 100;
  HelpWindow->Attr.W = 300;
  HelpWindow->Attr.H = 300;
  GetApplication()->MakeWindow(HelpWindow);
}
```

ウィンドウオブジェクトの作成については，148ページの「**ウィンドウオブジェクトの初期化**」を参照してください。

MakeWindow 関数

TModule オブジェクトには，***MakeWindow*** という重要なメンバ関数が定義されています。***MakeWindow*** は，インターフェース要素とウィンドウオブジェクトを安全に対応づけるためのすべての処理を実行します。「安全に」というのは，システムハングのような重大障害が発生しないうちに，***MakeWindow*** がエラー状態を検出するという意味です。***MakeWindow*** による処理には重要なステップが2つあります。

1. 最初のステップで，インターフェースオブジェクトの妥当性をテストします。このテストは ***ValidWindow*** を呼び出して行ないます。アプリケーションがメモリ不足にならなかったかどうか，またアプリケーション自体が正しく生成されたかどうかを調べることができます。
2. インターフェースオブジェクトが正常に作成されると，***MakeWindow*** はオブジェクトのメンバ関数 ***Create*** を呼び出してインターフェース要素の作成と対応づけを行なわせます。***Create*** は，オブジェクトのデータメンバ内の情報に基づいて，どのようなインターフェース要素を作成するかを Windows に指示します。要素が作成できない場合にはエラーメッセージボックスが現われます。

妥当性チェックに合格しなかった場合，またはインターフェース要素の作成に失敗した場合，***MakeWindow*** は無効なウィンドウオブジェクトを削除してヌルを返します。成功した場合は，引数として渡されたインターフェースオブジェクトのポインタを返します。

ダイアログボックスの追加

ダイアログボックスはポップアップウィンドウに似ていますが，通常は画面上に短時間しか表示されません。ダイアログボックスはプリンタの選択やドキュメントのページ設定といった特定の入力関連タスクに使われます。ここでは，ファイルのオープンとセーブ用のダイアログボックスを STEP アプリケーションに追加してみます。

ポップアップウィンドウと同様，ダイアログボックスも独立子ウィンドウです。ダイアログボックスを子ウィンドウとして追加するのは，ポップアップウィンドウを追加するのと原理的にはまったく同じです。両者はたいへんよく似ていますが，次のような違いがあります。

- ダイアログクラスは，***TWindow*** の下位クラスではなく ***TDialog*** の下位クラスです。***TWindow*** と ***TDialog*** はともに ***TWindowsObject*** の下位クラスです。
- 通常ダイアログボックスには，そのサイズ，位置，外観を指定する**リソース**が必要です。
- 通常ダイアログボックスは，簡単なタスクを実行して値を返します。たとえば，ステップ2で作成した ***CanClose*** メッセージボックスは機能が限られたダイアログであり，ユーザからの yes または no という応答を返すものでした。また，ステップ5で作成したダイアログボックスは線の太さを指定するものでした。

ここでは，ObjectWindows のストックダイアログボックスの1つであるファイルダイアログボックスを追加することにします。このダイアログボックスは，***TDialog*** の派生クラスである ***TFileDialog*** クラスで定義されています。ファイルダイアログボックスは，セーブやロードを行なうディスクファイルをユーザに指定させる場合など，広汎な用途が考えられます。たとえば，ワープロのアプリケーションでは，ファイルダイアログボックスを使ってドキュメントのオープンやセーブが行なえます。ユーザが **File** | **Open** か **File** | **SaveAs** を選択したときに，ファイルダイアログボックスを表示させるようにします。また，ファイルダイアログボックスは，「この機能はありません」というメッセージボックスのかわりになります。ステップ9では，実際のファイルに結合し，セーブやオープンを行なって，データの格納と検索を行なうことにします。いまのところはダイアログボックスを表示するだけにします。ファイルダイアログボックスの外観を**図 5.2** に示します。

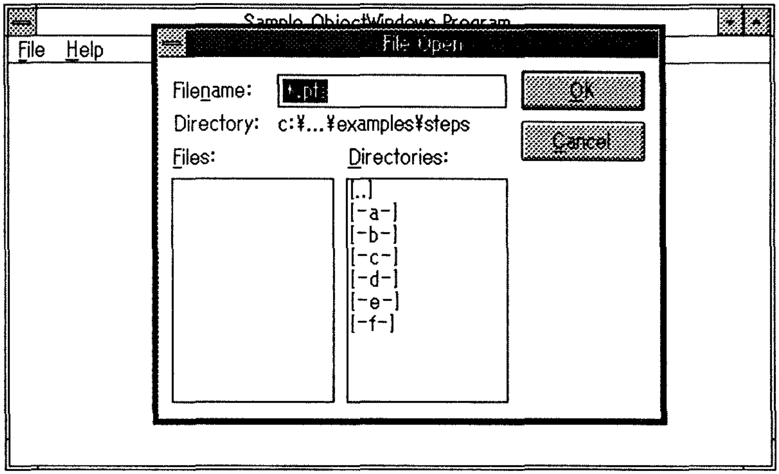


図5.2
ファイルダイアログボックスつきのSTEPアプリケーション

データメンバの追加

ファイルダイアログボックスオブジェクト全体を親ウィンドウのデータメンバとして格納せずに，必要に応じてそのつど新しいファイルダイアログボックスオブジェクトを構築することにします。この場合，格納するのは，ファイルダイアログボックスで使用するデータ，すなわちファイル名だけです。オブジェクト全体を必要もないのに保持しておくということは避け，オブジェクトの初期化に必要なデータだけを格納することにします。

ファイルダイアログボックスの構築には，親ウィンドウ，リソース識別子(ID)，ファイル名の3つの引数が必要です。リソースIDはオープン用またはセーブ用の標準ファイルダイアログボックスの識別子，すなわち，***SD_FILEOPEN***か***SD_FILESAVE***です。選択されたファイル名はファイル名引数で返されます。この引数でセーブ用ファイルダイアログのデフォルトのファイル名も渡されます。これはオープン用ファイルダイアログのデフォルトのファイルマスクを渡すのにも使います。

*TMyWindow*の宣言を次に示しておきます。

```
class TMyWindow : public TWindow
{
:
  char FileName[MAXPATH];
:
};
```

ダイアログボックスの実行

TFileDialog のコンストラクタに渡されたリソースIDに応じて，ダイアログボックスはファイルのオープンまたはセーブに使われます。どちらの場合でも，ダイアログボックスの外観は**図 5.2** とほとんど同じです。ただし，ファイルセーブダイアログでは，最初に現在のファイル名をダイアログのコンボボックスの編集領域に表示します。

CMFileOpen と ***CMFileSaveAs*** は次のように書き直すことができます。

このコードはSTEP 8.CPPの一部です。

```
void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILEOPEN, strcpy(FileName,
                                    "*.PTS"))) == IDOK )
     MessageBox(HWindow, FileName, "Open the file:", MB_OK);
}

void TMyWindow::CMFileSaveAs(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILESAVE, FileName)) == IDOK )
      MessageBox(HWindow, FileName, "Save the file:", MB_OK);
}
```

ここまでのSTEPアプリケーションのソースコードは，ファイルSTEP 8.CPPに収めてあります。

TApplication :: ***MakeWindow*** を使って「安全な」インターフェース要素を作成したように，***TApplication*** :: ***ExecDialog*** を使うと安全なダイアログボックスを作成して実行することができます。***MakeWindow*** と同様，***ExecDialog*** は ***ValidWindow*** を呼び出して，ダイアログボックスオブジェクトが作成されたかどうかを確認します。作成されていれば，ダイアログボックスオブジェクトのメンバ関数 ***Execute*** を呼び出します。ダイアログボックスの実行後にエラーが発生しなければ，***ExecDialog*** は ***Execute*** からの戻り値を返します。エラーが発生した場合は，***IDCANCEL*** を返して，ユーザがダイアログボックスを取り消したかのようにプログラムに通知します。どちらの場合も，***ExecDialog*** はダイアログボックスオブジェクトをメモリから削除します。

ExecDialog を使ってダイアログボックスを実行するのが最も安全な方法です。***Execute*** を直接呼び出すこともできますが，システムがハングアップする場合があるので危険です。

ステップ9：描画内容をファイルに格納する

以上で，現在の線のデータ表現を *TMyWindow*（データメンバ *Points*）に格納しました。あとはこのデータをファイル（実際にはファイルストリーム）に転送したり，読み出せるようにしなければなりません。このステップでは，状態情報を保持するデータメンバを追加するとともに，メンバ関数を変更して，.PTS ファイルのセーブとオープンを行なわせることにします。

状態の監視

描画では監視すべき点が2つあります。1つは，ファイルをセーブするかどうかという点です。もう1つは，現在ロードされているファイルがあるかどうかということです。この2つの条件は，*TMyWindow* の論理属性としてとらえることができます。したがって，次のようなデータメンバとして定義します。

```
class TMyWindow : public TWindow
{
:
  BOOL IsDirty, IsNewFile;
:
};
```

現在の描画内容が「ダーティ」であれば ***IsDirty*** は真になります。「ダーティ」とは，描画内容が前回のセーブ以後に変更されており，セーブが必要であるという意味です。または，一度もセーブされていないという意味です。ユーザが描画を始めた ***WMLButtonDown*** 時点で ***IsDirty*** を真に設定します。ユーザが新規ファイルをオープンするか既存ファイルをセーブしたら，***IsDirty*** を偽に設定します。ユーザがアプリケーションをクローズしたら（***CanClose***）***IsDirty*** をチェックし，真であればメッセージボックスを表示します。

ウィンドウが最初に表示されたときと，File | New メニュー（***CMFileNew***）が選択された直後だけ ***IsNewFile*** が真になります。ファイルがオープン（***CMFileOpen***）またはセーブ（***CMFileSave*** か ***CMFileSaveAs***）されると偽になります。***CMFileSave*** は ***IsNewFile*** を使って，ファイルをただちにセーブできるか（偽），またはユーザにファイル名をファイルダイアログから選択させるのか（真）を判断します。

メンバ関数 ***CanClose*** とファイルのセーブおよびロード用のメンバ関数を以下に示します。ここまでのプログラムは，ファイルのセーブとロード以外はすべて行なえます。ファイルのセーブは ***SaveFile*** という新しいメンバ関数にまとめてあり，ファイルのオープンは ***OpenFile*** で行なわれます。

このコードはSTEP 9.CPPの一部です。

```
BOOL TMyWindow::CanClose()
{
  if ( !IsDirty )
    return TRUE;
  return MessageBox(HWindow, "Do you want to save?", "Drawing has
                   changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}

void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  IsDirty = FALSE;
  IsNewFile = TRUE;
}
```

```
void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                  SD_FILEOPEN, strcpy(FileName,
                                  "*.PTS"))) == IDOK )

        OpenFile();
}

void TMyWindow::CMFileSave(RTMessage)
{
  if ( IsNewFile )
    SaveFileAs();
  else SaveFile();
}

void TMyWindow::SaveFileAs()
{
  if ( IsNewFile )
    strcpy(FileName, "");
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                  SD_FILESAVE, strcpy(FileName,
                                  "*.PTS"))) == IDOK )

      SaveFile();
}


void TMyWindow::CMFileSaveAs(RTMessage)
{
  SaveFileAs();
}

void TMyWindow::SaveFile()
{
   ofpstream os(FileName);

   os << Points;
   os.close();
   IsNewFile = IsDirty = FALSE;
}

void TMyWindow::OpenFile()
{
   ifpstream is(FileName);
   if ( is.bad() )
     MessageBox(HWindow, "Unable to open file", "File Error", MB_OK |
                 MB_ICONEXCLAMATION);
   else
   {
     Points->DeleteAll();
     is >> Points;
     is.close();
     IsNewFile = IsDirty = FALSE;
```

```
        InvalidateRect(HWindow, NULL, 1);
      }
    }
```

ファイルのオープンとセーブ

以上で.PTS ファイルのオープンとセーブ用の枠組みが構築されました。ただし，*Points* 配列の読み書きという実際の作業が残っています。この作業は *Points* を**ストリーム型**の配列にすると簡単です。

まず **Array** と ***TStreamable*** の両方から多重継承します。

このコード例は STEP 9 .CPP の一部です。

```
_CLASSDEF(TPointArray)
class TPointArray : public Array, public TStreamable
{
public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0)
    : Array(upper, lower, aDelta){};
:
protected:
  virtual void write( Ropstream );
  virtual Pvoid read( Ripstream );
:
};
```

TPointArray の読み出しまたは書き込みを要求すると，そのメンバ関数 *write* と *read* が呼び出されます。この 2 つの関数が *TPointArray* の座標位置群の読み出しと書き込みを実際に行ないます。以下に両関数のリストを示します。

このコード例は STEP 9 .CPP の一部です。

```
Pvoid TMyWindow::read(Ripstream is)
{
  TWindow::read(is);
  is >> Points;
  return this;
}

void TMyWindow::write(Ropstream os)
{
  TWindow::write(os);
  os << Points;
}
```

TPointArray を実際にストリームで使えるようにするには，他にいくつものメンバ関数を定義する必要がありますが，そのほんの一部の「ワンライナー」と呼ばれる単純なものが EXAMPLES¥STEPS サブディレクトリの STEP 9.CPP プログラムにあります。ストリーム型のメンバ関数の全貌については第 15 章で説明します。いまはとりあえず，メンバ関数の実現方法をざっと見る程度にしましょう。*TMyWindow* のメンバ関数 *read* と *write* を見れば，*TPointArray* がストリーム型のとき *TMyWindow* をストリーム型にするのがいかに簡単かがわかるはずです。

第6章

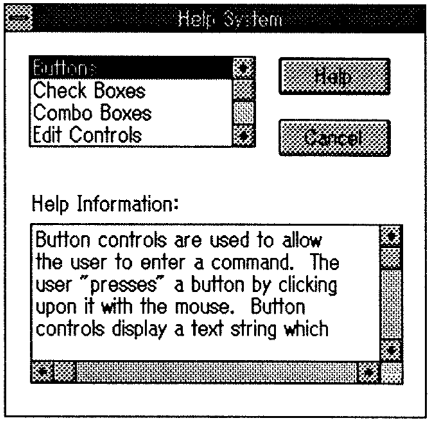
ウィンドウのポップアップ

最後に，ヘルプシステムを追加してObjectWindowsプログラムを完成させましょう。ヘルプシステムとは，プログラムの中にある即座に呼び出せるプログラムと考えてください。Windowsプログラムの長所の1つとして，複数のウィンドウを使いながら，多数の処理を同時に行なって統合できるという点を挙げることができます。ここでは，ヘルプシステムをSTEPアプリケーションに追加する作業を通じて，ユニットを使ったモジュール形式のObjectWindowsプログラムの設計方法を学んでいくことにします。この形式で設計すると，プログラムの再利用と保守性が飛躍的に向上します。

ステップ10：ヘルプウィンドウのポップアップ

ステップ8で，HelpメニューをSTEPアプリケーションのメインウィンドウに追加しました。このメニューを選ぶと，***TWindow***のインスタンスである空白のウィンドウが表示されました。これから，この空白のウィンドウを実用的なHelpウィンドウに取り換え，ObjectWindowsでサポートされているWindowsの各種コントロールを入れることにします。コントロールとして，リストボックス，スクロールバー，エディットコントロール，スタティックコントロール，ボタン，コンボボックスがあります。ヘルプシステムは独立したモジュールで定義されるので，既存のプログラムに手を加える必要はほとんどありません。完成したヘルプシステムを**図6.1**に示します。ObjectWindowsでサポートしているコントロールのいくつかが簡単に説明されています。

図 6.1
STEP アプリケーションのヘルプシステム



ObjectWindows でモジュールを使う

モジュールは，クラスのメンバ関数の定義を記録しておくための便利な仕組みです。モジュールの使用は，Windows のプログラミングでは特に重要になります。Windows ではウィンドウ（ObjectWindows のウィンドウオブジェクト）をあるアプリケーションから別のアプリケーションへと切り換えることが多いためです。ここでは，ヘルプシステムを *THelpWindow* というクラスとして構築し，その定義を HELPWIND.CPP というファイルに格納することにします。

メインプログラムの変更

STEP アプリケーションに必要な変更は次の 2 つだけです。

- HELPWIND.CPP モジュールを STEP アプリケーションプロジェクトかメイクファイルに追加する。

- HELPWIND.CPP の属性を設定するコード全体を *TMyWindow* :: *CMHelp* から削除し，*THelpWindow* のコンストラクタに入れる。

これで，ユーザがメニュー項目 **Help** を選択すると，STEP アプリケーションでは *THelpWindow* を生成します。したがって，*THelpWindow* クラスを定義しておかなければなりません。このクラスは HELPWIND.CPP モジュールで定義します。

モジュールの作成

HelpWind モジュールは，*THelpWindow* クラスのインターフェース部と実現部の定義しか行ないません。このモジュールをいったん作成すると，どの ObjectWindows プログラムでも使うことができます。事実上モジュール内のウィンドウということになります。*HelpWind* モジュールの形式を以下に示します。空白部分は後で完成させます。

このコードは HELPWIND.H にあります。

```
#ifndef __HELPWIND_H
#define __HELPWIND_H

#ifndef __OWL_H
#include <owl.h>
#endif

#ifndef __LISTBOX_H
#include <listbox.h>
#endif

#ifndef __EDIT_H
#include <edit.h>
#endif

#define ID_LISTBOX  101
#define ID_BUTTON1  102
#define ID_BUTTON2  103
#define ID_EDIT     104

_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
```

```
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};

#endif
```

このコードはHELPWIND.CPPにあります。

```
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <owl.h>
#include <static.h>
#include <edit.h>
#include <listbox.h>
#include <button.h>
#include "helpwind.h"

THelpWindow::THelpWindow(PTWindowsObject AParent) :
   TWindow(AParent, "Help System")
{
⋮
}

void THelpWindow::SetupWindow()
{
⋮
}

void THelpWindow::HandleListBoxMsg(RTMessage Msg)
{
⋮
}

void THelpWindow::HandleButton1Msg(RTMessage)
{
⋮
}

void THelpWindow::HandleButton2Msg(RTMessage)
{
⋮
}

void THelpWindow::FillEdit(Pchar SelString)
{
⋮
}
```

ウィンドウにコントロールを追加する

ステップ8で，子ウィンドウには従属と独立の2種類があることを学び，メインウィンドウにダイアログとポップアップウィンドウという2種類の独立子ウィンドウを追加しました。ここでは，従属子ウィンドウの一種である**コントロール**を追加することにします。Helpウィンドウをこのコントロールの親ウィンドウにすることにします。Helpウィンドウは独立子ウィンドウで，その親ウィンドウはアプリケーションのメインウィンドウです。したがって，ヘルプウィンドウは同時に子ウィンドウでも親ウィンドウでもあるわけです。

コントロールの実体

コントロールとは，ウィンドウまたはダイアログボックスのユーザインターフェースの一部または全体を構成する視覚的な部品です。たとえば，ファイルダイアログボックスのOKボタンやCancelボタンはコントロールです。ファイルを表示するためのコンボボックスもコントロールです。STEPアプリケーションのメインウィンドウにはコントロールがありませんが，ここでHelpウィンドウにいくつか追加することにします。

コントロールにはリストボックス，スクロールバー，エディットコントロール，スタティックコントロール，ボタン，コンボボックスというように多くの種類があります。**図6.1**のHelpウィンドウには，リストボックス（ウィンドウの上半分のボックス），エディットコントロール（ウィンドウの下半分のボックス），スタティックコントロール（“Help Information：”というテキスト），2つのボタン（HelpとCancel）が使われています。スクロールバーは，エディットコントロールとリストボックスに組み込まれていますが，スクロールバーオブジェクトではありません。スクロールバーは他のコントロールやウィンドウのオプション部品としてよく使われますが，独立したコントロールとしてスクロールバーだけで使われることはまれです。

ObjectWindowsの ***TControl*** クラスは，すべてのコントロールウィンドウオブジェクトに共通な動作を規定します。その派生クラスは，各インスタンスが表わす各種のコントロールウィンドウに応じた固有の動作を定義します。たとえば，***TEdit*** はエディットコントロール固有の動作を，***TListBox*** はリストボックス固有の動作を定義します。また，***TControl*** が ***TWindow*** の下位クラスであるという点にも注意が必要です。つまり，コントロールは特殊な形態のウィンドウなのです。

ウィンドウコントロールの作成

リソース定義のコントロールと ObjectWindows オブジェクトを対応づけることができます。

まったく同じように動作するコントロールでも，ダイアログボックス（ファイルダイアログ）のコントロールとウィンドウ（ヘルプウィンドウなど）のコントロールでは，プログラミングが大きく異なります。ダイアログボックスのコントロールの特性は，ダイアログのリソース定義で指定されます。Windowsは，このリソース定義を使ってダイアログのコントロールを作成します。ダイアログコントロールはObjectWindowsのオブジェクトに対応していないこともあります。ダイアログのコントロールがObjectWindowsオブジェクトに対応していない場合は，ダイアログ自身でWindows関数を直接呼び出してコントロールを管理しなければなりません。

他方，ウィンドウのコントロールの場合は常にObjectWindowsオブジェクトとして定義されます。親ウィンドウは，ObjectWindowsのコントロールオブジェクトによって定義されている豊富なメンバ関数群を使ってコントロールを管理します。たとえば，ユーザがリストボックスから選択したテキスト項目を取得するには，リストボックスオブジェクトのメンバ関数 ***GetSelString*** を呼び出します。ウィンドウオブジェクトやダイアログボックスオブジェクトと同様，コントロールオブジェクトにも対応する視覚的要素があります。ウィンドウとダイアログの場合のように，コントロールオブジェクトとコントロール要素は，コントロールオブジェクトのデータメンバ ***HWindow*** によって互いにリンクされています。また，それぞれのコントロールには固有のIDがあり，親ウィンドウはコントロールイベントを送るとき，このIDでコントロールを識別します。たとえば，ユーザがボタンをクリックすると，親ウィンドウはこのIDでボタンを識別します。以下のように各コントロールIDに定数を定義すると明確になります。

```
#define ID_LISTBOX 101;          // リストボックス1のID
#define ID_BUTTON1 102;          // 最初のボタン (OK) のID
#define ID_BUTTON2 103;          // 2番目のボタン(Cancel)のID
#define ID_EDIT 104;             // エディットコントロールのID
```

データメンバとしてのコントロールオブジェクト

コントロールオブジェクト（または他の子ウィンドウ）を指すポインタを，ウィンドウオブジェクトのデータメンバとして格納しておくと便利な場合があります。後でオブジェクトのメンバ関数を呼び出して直接処理しなければならない子ウィンドウの場合です。この対象は，リストボックスとエディットコントロールです。*THelpWindow* は，それぞれのコントロールオブジェクトを別々のデータメンバに格納します。次に示すのは *THelpWindow* のオブジェクト定義の一部です。

このコードはHELPWIND.Hにあります。

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  :
};
```

子ウィンドウのコントロールオブジェクトのインスタンスをいったん構築してしまえば，メンバ関数を呼び出して操作することができます。たとえば，***ListBox*** **->*****AddString*** を呼び出して *ListBox* のリストに文字列を追加することができます。***TWindowsObject*** のメンバ関数 ***Next*** と ***Previous*** を使えば，データメンバとして格納されていない子ウィンドウのコントロールオブジェクトにアクセスできますが，データメンバを使ったほうがはるかに簡単です。

コントロールの管理

コントロールオブジェクトを持つウィンドウクラスは，すべてコンストラクタを定義してコントロールオブジェクトを構築しなければなりません。***TWindow*** から継承した ***SetupWindow*** を定義し直してコントロールオブジェクトを設定することもできます。*THelpWindow* は ***SetupWindow*** を定義し直して，リストをセットアップし，子リストボックスの選択内容を設定します。なお，*THelpWindow* :: *SetupWindow* は最初に ***TWindow*** :: ***SetupWindow*** を呼び出し，これはさらに ***TWindowsObject*** :: ***SetupWindow*** を呼び出します。この ***TWindowsObject*** :: ***SetupWindow*** が *THelpWindow* の子ウィンドウとして構築されているウィンドウを作成します。

Help ウィンドウのコンストラクタのコードを以下に示します。コンスタラクタが最初に呼び出すのは ***DisableAutoCreate*** です。これは，作成するポップアップウィンドウのコンストラクタで呼び出されなければならない場合もあります。この呼び出しによって，ポップアップウィンドウの作成が親ウィンドウの作成とは無関係に行なわれることを示すフラグが設定されます。これは通常ポップアップにそぐわない動作といえます。

次に *THelpWindow* のコンストラクタがヘルプウィンドウの位置属性とサイズ属性を設定します。*THelpWindow* は子ウィンドウなしでは充分な機能を発揮しないので，*THelpWindow* の子ウィンドウが構築されるまでは完全に構築されたとはいえません。*THelpWindow* は this を第 1 引数としてそれぞれのコントロールのコンストラクタに渡します（this は構築する *THelpWindow* オブジェクトを指すポインタです)。第 2 引数としてコントロール ID が各コントロールのコンストラクタ呼出しで渡されます。他の引数としては，位置データとサイズデータがあります。なお，*Listbox* と *Edit* は後でそれぞれのコントロールが参照されるときのために設定されます。

このコード例は **HELPWIND.CPP** の一部です。

```
THelpWindow::THelpWindow(PTWindowsObject AParent) :
   TWindow(AParent, "Help System")
{
  DisableAutoCreate();
  Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
```

```
  Attr.H = 300;
  ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 180, 80);
  new TButton(this, ID_BUTTON1, "Help", 220, 20, 60, 30, TRUE);
  new TButton(this, ID_BUTTON2, "Cancel", 220, 70, 60, 30, FALSE);
  Edit = new TEdit(this, ID_EDIT, "", 20, 180, 260, 90, 40, TRUE);
  new TStatic(this, -1, "Help Information:", 20, 160, 160, 20, 0);
}
```

ヘルプウィンドウを作成したら，*THelpWindow::SetupWindow* を呼び出してヘルプウィンドウをセットアップします。ヘルプウィンドウのコントロールを作成し，リストボックスを初期化するとヘルプウィンドウのセットアップが完了します。

THelpWindow::SetupWindow はまず ***TWindow::SetupWindow*** を呼び出してヘルプウィンドウの子ウィンドウを作成させます。次に *ListBox* のメンバ関数を呼び出して，リストとその選択内容を初期化させます。なお，***WS_VISIBLE*** が作成スタイルとしてデフォルトで設定されるので，コントロールは作成と同時に自動的に表示されます。

このコード例は HELPWIND.CPP の一部です。

```
void THelpWindow::SetupWindow()
{
  TWindow::SetupWindow();
  ListBox->AddString("List Boxes");
  ListBox->AddString("Buttons");
  ListBox->AddString("Check Boxes");
  ListBox->AddString("Radio Buttons");
  ListBox->AddString("Group Boxes");
  ListBox->AddString("Scroll Bars");
  ListBox->AddString("Edit Controls");
  ListBox->AddString("Static Controls");
  ListBox->AddString("Combo Boxes");
  ListBox->SetSelIndex(0);
};
```

THelpWindow のコンストラクタとメンバ関数 ***SetupWindow*** を呼び出すだけで，Help ウィンドウのコントロールを正しく表示させることができます。リストボックスはスクロールすることができ，ボタンも押すことができます。ただし，何の応答も起きません。次の節ではコントロールイベントに対する応答を定義することにします。

コントロールイベントへの応答

メニューイベントに応答するのに，コマンドメッセージ応答メンバ関数を定義しましたが，コントロールイベントに応答するには，子IDメッセージ応答メンバ関数を定義します。子IDメッセージ応答メンバ関数を識別するには，次のように定数 ***ID_FIRST*** とコントロールのIDの和を使います。

このコードはHELPWIND.Hの一部です。

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
:
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
:
};
```

ID_FIRST は ObjectWindows で定義されている定数です。上の例では，***ID_LISTBOX*** が *TMyWindow* を親とするリストボックスコントロールの識別子になっています。*HandleListBoxMsg* が，このリストボックスコントロールから送られるすべてのメッセージに対する応答を定義します。

親に知らせなければならないイベントが起こるたびに，コントロールから親にメッセージが送られます。親にコントロール通知メッセージが送られるイベントは，子コントロールの種類によって異なります。発生したイベントの種類を識別する通知コードがこのメッセージで親に渡されます。たとえば，リストボックスの選択内容が変わるとリストボックスから通知メッセージ ***LBN_SELCHANGE*** が送られます。ボタンがクリックされるとボタンからメッセージ ***BN_CLICKED*** が送られます。

コントロール応答メンバ関数には，コマンドメッセージ応答メンバ関数のように ***RTMessage*** 型の引数 *Msg* が渡されます。*Msg.LP.Hi* には発生したイベントの種類を示すコードが入り，*Msg.LP.Lo* にはコントロールのウィンドウハンドルが入ります。*Msg.WParam* にはコントロールのIDが入ります。

コントロール通知メッセージで送られるコードの一覧は **ObjectWindows for C++リファレンスガイド**にあります。メッセージ応答メンバ関数には，*HandleListBoxMsg* のようなわかりやすい名前をつけてください。メンバ関数のプロトタイプを示す *THelpWindow* のクラス定義を以下に示します。

このコードはHELPWIND.Hの一部です。

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};
```

次に，コントロールからのメッセージに対して ***THelpWindow*** をどのように応答させるかを決めます。ユーザがCancelボタンをクリックしたら（***ID_BUTTON 2,***）ウィンドウをクローズすることにします。この場合，*HandleButton 2 Msg* の実現は次のように大変簡単です。

```
void THelpWindow::HandleButton2Msg(RTMessage Msg)
{
  CloseWindow();
}
```

ボタンから送られるコントロール通知メッセージは，***BN_CLICKED*** と ***BN_DBLCLICKED*** の2種類のみです。どちらの場合もウィンドウはクローズするので，受信したコントロールメッセージの通知コードをチェックする必要はありません。

ユーザがOKボタン ***ID_BUTTON 1*** をクリックしたら，STEPアプリケーションにどのリストボックス項目が選択されたかをチェックさせ，対応するテキストをエディットコントロール ***ID_EDIT*** に書き込ませることにします。この場合，*HandleButton 1 Msg* でリストボックス ***ID_LISTBOX*** の ***GetSelString*** を呼び出し，選択されたリスト項目のテキストを取得します。このテキストは，後で記述する *FillEdit* というメンバ関数に渡されます。*FillEdit* の定義はHELPWIND.CPPにあります。

```
void THelpWindow::HandleButton1Msg(RTMessage Msg) {
  char SelString[25];

  ListBox->GetSelString(SelString, sizeof (SelString));
  FillEdit(SelString);
}
```

ユーザがリストボックス項目を直接ダブルクリックしたときにも，エディットコントロールにテキストを表示させることができます。ただし，リストボックスからは各種のメッセージが親に送られるので，エディットコントロールに書き込む前にメッセージのコードを調べる必要があります。コードの値が ***LBN_DBLCLK*** であれば，ユーザがダブルクリックしています。リストボックスに送られるその他のメッセージはすべて ***DefWndProc*** を呼び出して処理できます。

DefWndProc の詳細については 144 ページを参照してください。

```
void THelpWindow::HandleListBoxMsg(RTMessage Msg) {
  char SelString[25];

  if (Msg.LP.Hi == LBN_DBLCLK)
  {
    ListBox->GetSelString(SelString, sizeof (SelString));
    FillEdit(SelString);
  }
  else DefWndProc(Msg);
}
```

ObjectWindows のコントロールオブジェクトには，各種の照会や変更用のメンバ関数が提供されています。コントロールイベントに応じたメンバ関数を呼び出すことで，Help ウィンドウの場合と同様，ウィンドウを状況に応じて変化させることができます。

PART 2

第2部

ObjectWindowsを使う

第7章

ObjectWindowsの概要

ObjectWindowsはクラスの集合体であり，C++によるMicrosoft Windowsプログラムの開発を円滑にするためのものです。

この章では，ObjectWindowsのクラス階層全体を概説します。残りの章では，それぞれの階層の詳細について説明します。

オブジェクト階層の説明だけでなく，リソースの使用，Windows APIの呼び出し，Windowsからのメッセージの受理と処理といったことも含め，Windows環境でのプログラミングの基本概念についてもこの章で言及します。

ObjectWindowsの規約

ObjectWindows階層内のクラスと構造体の名前は多くの場合，***TWindow***のように*T*で始まります。ObjectWindowsでは，*T*で始まるすべてのクラスに対し以下の関連型も定義しています。

- ***Pclass***。クラス*class*を指すポインタ型（例：***TWindow***を指すポインタとして定義される***PTWindow***）。
- ***Rclass***。クラス*class*への参照型（例：***TWindow***への参照として定義される***RTWindow***）。
- ***RPclass***。クラス*class*を指すポインタへの参照（例：***TWindow***を指すポインタの参照としてとして定義される***RPTWindow***）。
- ***PCclass***。const クラス*class*を指すポインタ型（例：const ***TWindow***を指すポインタとして定義される***PCTWindow***）。
- ***RCclass***。const クラス*class*への参照型（例：const ***TWindow***への参照として定義される***RCTWindow***）。

以上の型は，Windowsの複雑なプログラミング手続きを簡単にするものです。以下のダイナミックリンクライブラリ（DLL）の宣言と比較してみてください。

これはPTWindowに対応します。

```
TWindow _FAR *
```

DLLでは，上のように ***TWindow*** を指すポインタや，

これはRPTWindowに対応します。

```
TWindow _FAR * _FAR &
```

上のように ***TWindow*** を指すポインタへの参照が必要でした。

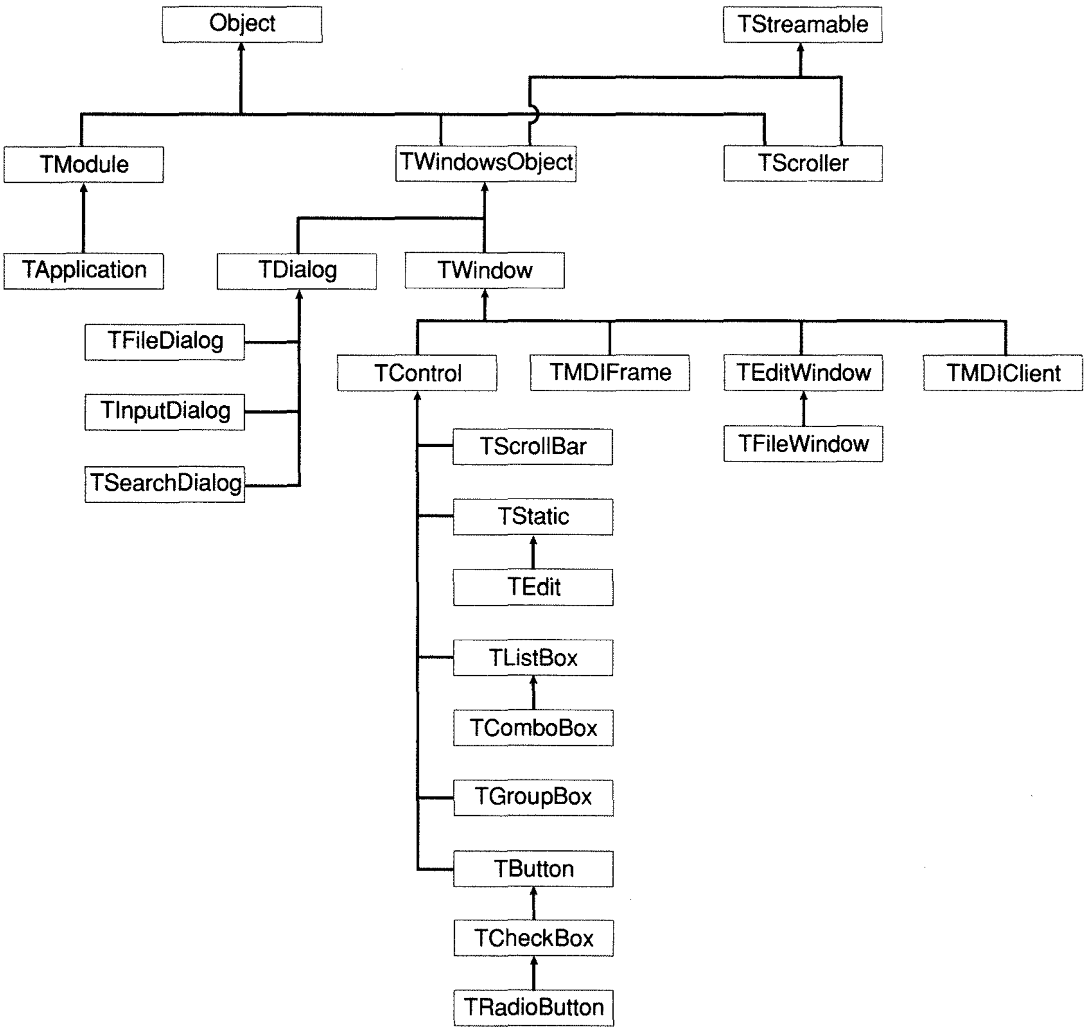
P，***R***，***RP***，***PC***，***RC*** の各型を使うと，一部を後でDLLにする場合に，アプリケーションを簡単に変換できます。ObjectWindowsのサンプルプログラムとSTEPアプリケーションでは，これらの型を随所に使っています。

メッセージ応答メンバ関数は，応答の対象となるメッセージにちなんだ名前とし，メッセージ名から下線を取り除いたものにします。たとえば，***WM_KEYDOWN*** メッセージに応答するメンバ関数名は，***WMKeyDown*** となります。また，***CM_FILEOPEN*** コマンドに応答するメッセージは，***CMFileOpen*** となります。

ObjectWindowsの階層

ObjectWindowsは，クラスを階層状にまとめたライブラリです。継承を用いることによって，変更や追加などが自由に行なえます。ライブラリ内のクラス，およびそのデータメンバとメンバ関数の詳細については，**ObjectWindows for C++リファレンスガイド**の第1章を参照してください。

図 7.1
ObjectWindows のクラス階層



Object

Object は ObjectWindows のすべての派生クラスの基本クラスであり，コンテナクラスライブラリで定義されています。コンテナクラスライブラリでは名前の語頭に *T* がつかないので，***Object*** 名は ***TObject*** になりません。

TModule

ObjectWindows のダイナミックリンクライブラリ（DLL）が ***TModule*** のインスタンスを構築します。このインスタンスはライブラリ（DLL）モジュールの「代役」として動作します。ObjectWindows アプリケーションも ***TModule*** から派生させた ***TApplication*** のインスタンスを構築します。***TModule*** はライブラリモジュールとアプリケーションモジュールに共通な動作を定義するわけです。さらに，***TModule*** のメンバ関数ではウィンドウメモリ管理とエラー処理もできます。

TApplication

このクラスは，すべての ObjectWindows アプリケーションに必要な動作を定義します。新たに作成する ObjectWindows アプリケーションは，***TApplication*** の派生クラスになります。***TApplication*** の一番重要な役割はメインウィンドウオブジェクトの初期化です。アプリケーションオブジェクトに関しては，第 8 章『**モジュールとアプリケーションオブジェクト**』に詳細な説明があります。

インターフェースオブジェクト

ObjectWindows 階層におけるその他のオブジェクトは，一般に**インターフェースオブジェクト**として分類されます。その理由は次の 2 つです。第 1 は，これらのオブジェクトが，Windows ユーザインターフェースにおける要素を表わすものだからです。第 2 は，アプリケーションの各部分と Windows 環境の間の一種のインターフェースになっているためです。インターフェースオブジェクトに関しては，第 9 章『**インターフェースオブジェクト**』に詳細な説明があります。

TWindowsObject

TWindowsObject は基本クラスで，ウィンドウ，ダイアログボックス，コントロールという 3 種類の ObjectWindows インターフェースオブジェクをひとまとめにしたものです。この基本クラスでは，ウィンドウオブジェクトの作成，メッセージ処理，ウィンドウオブジェクトの廃棄といった処理のためのメンバ関数が提供されます。

ウィンドウオブジェクト

ウィンドウオブジェクトは，ウィンドウ環境のウィンドウだけではなく，この環境内で使われるコントロールのような大部分の視覚的要素も表わします。

TWindow

TWindow は汎用的なウィンドウクラスで，そのインスタンスでアプリケーションのメインウィンドウ，ポップアップウィンドウ，子ウィンドウを表わすことができます。***TWindow*** のインスタンスでもグラフィックスを表示できますが，ほとんどの場合，***TWindow*** からの派生クラスで ***TWindow*** の動作を専用化することになります。

TEditWindow

TWindow からの派生クラスで，ウィンドウでのテキスト編集を可能にするクラスを定義します。

TFileWindow

TEditWindow からの派生クラスで，ウィンドウでのテキスト編集ばかりでなく，テキストファイルのロードとセーブを行なえるクラスを定義します。

ダイアログオブジェクト

ダイアログオブジェクトによって，対話処理を簡単にするためのボタン，リストボックス，スクロールバーのようなコントロール群を作成できます。ダイアログオブジェクトに関しては，第 11 章『**ダイアログオブジェクト**』に詳細な説明があります。

TDialog

モードつきダイアログボックスとモードなしダイアログボックスの違いについては 174 ページを参照してください。

このクラスから派生させたクラスでは，Windows ダイアログボックスを管理することができます。ダイアログオブジェクトはダイアログリソースに対応しており，モードつきダイアログボックスとしてでもモードなしのダイアログボックスとしてでも実行できます。ダイアログとそのコントロールの間の通信を管理するためのメンバ関数も提供されます。

TFileDialog

多くのアプリケーションですぐに役に立つダイアログクラスです。オープン，編集，セーブの対象とするファイルをユーザに選択させるためのダイアログを定義します。

TInputDialog

ユーザに単一のテキスト項目を入力させるためのダイアログボックスを定義します。

コントロールオブジェクト

ダイアログやウィンドウ内でコントロールを使うと，ユーザにデータの入力やオプションの選択をさせることができます。コントロールオブジェクトによって，Windows で定義される様々な種類のコントロールを首尾一貫した簡潔なやり方で処理できます。コントロールオブジェクトに関しては，第 12 章『**コントロールオブジェクト**』に詳細な説明があります。

TControl

コントロールの例については図 12.1 を参照してください。

リストボックスやボタンなどすべてのコントロールオブジェクトに共通な基本クラスとして使われる抽象クラスです。派生クラスのコントロールを作成し，メッセージを処理するメンバ関数を定義しています。

TButton

Windows のプッシュボタンを表わす ***TButton*** のインスタンスです。

TCheckBox

TButton からの派生クラスです。このインスタンスは Windows のチェックボックスを表わし，その状態を管理するメンバ関数を提供します。

TRadioButton

TCheckBox からの派生クラスです。このインスタンスは Windows のラジオボタンの作成と状態管理を行ないます。

TListBox

このインスタンスは Windows のリストボックスを表わします。このクラスは Windows のリストボックスの作成とリストボックスからの選択を処理します。また，リスト内の項目を操作するメンバ関数を定義します。

TComboBox

TListBox からの派生クラスで，Windows のコンボボックスの動作を定義します。コンボボックスとは，リストボックスとエディットコントロールを組み合せたものです。

TGroupBox

このインスタンスは Windows のグループボックスを表わします。

TStatic

スタティック（表示のみという意味）コントロールで使用するテキストの設定，問い合わせ，クリアを行なうためのメンバ関数を提供します。

TEdit

TStatic からの派生クラスです。Windows のエディットコントロールに豊富なテキスト処理機能を提供します。

TScrollBar

独立したスクロールバーコントロールの範囲とスクロールボックス位置を管理するためのメンバ関数を定義します。

MDI オブジェクト

Windows は，単一のウィンドウ枠内で複数のウィンドウを処理するための標準方式を実現しています。この標準方式はマルチドキュメントインターフェース（MDI : Multiple Document Interface）と呼ばれます。ObjectWindows は MDI ウィンドウの設定および処理に必要な手段を提供します。MDI オブジェクトに関しては，第 14 章『**MDI オブジェクト**』に詳細な説明があります。

TMDIFrame

Windows の MDI 仕様に準拠したアプリケーションにおいて，そのメインウィンドウにふさわしいウィンドウ処理の動作を提供します。

TMDIClient

MDI ウィンドウに対するサポート機能が追加されます。MDI クライアントオブジェクトとは，MDI ウィンドウのクライアント領域を実際に管理するオブジェクトのことです。また，クライアント領域とは，MDI ウィンドウ(「ドキュメント」)を含む領域のことをいいます。

スクローラオブジェクト

TScroller はウィンドウのスクロールバーを動かす ObjectWindows オブジェクトです。これによって，ウィンドウに書き込んだテキストやグラフィックスを自動的にスクロールすることができます。ユーザがウィンドウのクライアント領域の内側から外側にマウスをドラッグすると，オーナーウィンドウもスクロールします。したがって，***TScroller*** はスクロールバーのないウィンドウにも使えるわけです。

Windows API 関数

Windows APIには，約600個の関数があります。Windowsアプリケーションは Windows関数を呼び出して環境を操作し，外観を変更し，ユーザ入力に応答します。ところが，ObjectWindowsを使えば，ほとんどのWindows関数を直接呼び出さずに，ウィンドウの作成，ダイアログボックスの表示，コントロールの操作を行なうことができます。

ObjectWindowsによるWindows関数の呼び出し

ObjectWindowsの多くのメンバ関数でWindows関数が呼び出されます。とはいってもObjectWindowsが機能を複製するわけではなく，Windows関数のいくつかをオブジェクト指向のアプリケーションの枠組にまとめ直すのです。さらに，ObjectWindowsを使えば，Windows関数が必要とする多数の引数を指定するという手間が大幅に省けます。多くの場合，ObjectWindowsはウィンドウハンドルや子ウィンドウなどの引数を自動的に指定するからです。これらの引数はインターフェースオブジェクトのデータメンバとして格納されています。

たとえば，多くのWindows関数は，処理対象のウィンドウを指すハンドルを必要とします。通常，これらの関数は，ウィンドウオブジェクトのメンバ関数から呼び出されます。ウィンドウオブジェクトは，対応するウィンドウのハンドルをデータメンバ ***HWindow*** に保持しているのでハンドルとして渡すことができます。したがって，同項目をそのつど指定する必要がなくなります。このウィンドウハンドル（およびWindowsが必要とするその他の多数の情報ビット）はオブジェクト内で**カプセル化**されているので，ObjectWindowsのクラスを，オブジェクト指向でないWindows APIに対するオブジェクト指向のインターフェースとして使えます。

Windows関数へのアクセス

WindowsのAPI関数はすべてObjectWindowsのプログラムでも使えます(関数一覧については**ObjectWindows for C++リファレンスガイド**の第4章『**Windows関数**』，またはファイルWINDOWS.Hを参照)。たとえば，以下のコ

ードは Windows 関数 ***MessageBox*** を呼び出してメッセージボックスを作成させるものです。

```
Reply = MessageBox(HWindow, "Do you want to save?",
                   "File has changed", MB_YESNO | MB_ICONQUESTION);
```

多くの Windows 関数には戻り値がありますが，***MessageBox*** の場合であれば，上の例で ***Reply*** に格納される整数が戻り値になります。この整数値はユーザがメッセージボックスをクローズしたときの動作を示します。ユーザが Yes ボタンをクリックした場合，Windows で定義済みの定数 ***IDYES*** と同じ値になります。No ボタンをクリックした場合は ***IDNO*** と同じ値になります。

Windows 関数では，各種の ***WORD*** 定数と ***DWORD*** 定数を引数として渡す必要があります。定数は，種々のスタイル，戻り値，識別子などを表わします。定数を使ってプログラムを記述すると，読みやすくなるだけでなく，保守性も向上し，さらに将来 Windows バージョンが更新されても何の影響も受けません。たとえばメッセージボックスを定義するとき，0x0024 とするより MB_YESNO | MB_ICONQUESTION とするほうがわかりやすくなります。***MB_YESNO***, ***IDYES*** などの定数は WINDOWS.H に定義してあります。

WINDOWS.H では，関数 ***MessageBox*** は以下のように定義されています。

```
int  FAR PASCAL MessageBox(HWND, LPSTR, LPSTR, WORD);
```

アプリケーションに自分で WINDOWS.H をインクルードする必要はありません。OWL.H ファイルが自動的にインクルードしてくれます。

int という語はすでにお馴染みのことと思います。しかし，***FAR***，***PASCAL***, ***HWND***，***LPSTR***，***WORD*** はどうでしょうか。いずれも WINDOWS.H に定義されている新しい型です。***FAR*** と ***PASCAL*** は far と pascal と同じものです（データ型 near, void, long の大文字についても WINDOWS.H に定義してあります)。***FAR*** cdecl 型に属する ***wsprintf*** を除き，Windows API 関数はすべて ***FAR PASCAL*** 型関数です。

関数の中にはもっと複雑なデータ構造体を要する，***LOGFONT*** などのフォントを記述するもの，***WNDCLASS*** などのウィンドウを記述するものもあります。このようなデータ構造体を必要とする Windows 関数には，通常，ポインタを引数として渡します。利用できる構造体の一覧は **ObjectWindows for C++リファレンスガイド**にあります。

スタイル定数の組み合せ

インターフェース要素を作成するWindows関数は，通常 ***WORD*** 型または ***DWORD*** 型のスタイル引数が必要です。Windowsでは数百個のスタイル定数が定義されています（**ObjectWindows for C++リファレンスガイド**の第7章『**Windows 定数**』に一覧があります)。スタイル定数識別子は，2文字のプレフィクスで始まり，その後に下線と内容に応じた名前が続きます。たとえば，***WS_POPUP*** というのは，ポップアップウィンドウのためのウィンドウスタイル定数です。このとき，***WS_*** によって，ウィンドウスタイルであることがわかります。

多くの場合，複数のスタイルをORビット演算子（|）で組み合せて別のスタイルを作成します。上記の ***MessageBox*** の例では，***MB_YESNO | MB_ICONQUESTION*** をスタイル引数として渡します。このスタイルでは，YesとNoの2つのボタンと疑問符のアイコンのついたメッセージボックスが作成されます。

スタイルによっては，相互に排他的なものがあるということに注意してください。このようなスタイルを組み合わせると，結果は予測できなくなり，おかしなものになります。

Windows 関数の種類

この節では，ObjectWindowsプログラムで利用できるWindows関数の種類を示します。

ウィンドウマネージャインターフェース関数

メッセージを処理し，ウィンドウとダイアログボックスを操作し，システム出力を作成します。この種の関数は，メニュー，カーソル，クリップボードを対象とします。

グラフィックスデバイスインターフェース（GDI）関数

画面やプリンタなど各種のデバイスにテキスト，グラフィックス，ビットマップを出力します。この種の関数は特定のデバイスと結び付いているわけではなく，デバイスに依存しないものです。

システムサービスインターフェース関数

メモリ管理，OS とのインターフェース，リソース管理，通信といった広範囲なシステムサービスを処理します。

コールバック関数

コールバック関数はユーザ指定の関数であり，プログラム外部のルーチンから呼び出されます。Windows 内で管理しているオブジェクト（フォントやウィンドウなど）の列挙が要求されると，Windows は通常コールバック関数を呼び出します。このような要求を Windows に対して行なう場合は，コールバック関数のアドレスを渡しておかなければなりません。Windows が要求者をコールバック関数で「呼び戻せる」ようにするためです。

すでに述べたように，コールバック関数はプログラムとは異なるコンテキストで呼び出されます。通常は Windows 内から呼び出されます。したがって，コールバック関数からプログラム内のデータセグメントにある定義済みのグローバル変数をアクセスするためには，そのデータセグメントがプログラムに対応するように正しく設定されていることを確認する手段が必要となります。Windows では「インスタンスサンク」という方法を用います。この方法ではコールバック関数の実際のアドレスを Windows に渡すのではなく，インスタンスサンクのアドレスを渡します。インスタンスサンクはデータセグメントレジスタにデータセグメントを設定すると，制御をコールバック関数に渡してしまいます。インスタンスサンクは，Windows 関数 ***MakeProcInstance*** によりコールバック関数ごとに作成されます。

```
FARPROC ThunkPtr = MakeProcInstance((FARPROC) MyCallback,
                                    GetApplication()->hInstance);
```

MakeProcInstance に渡される第1引数はコールバック関数を指すポインタです。第2引数はアプリケーションのインスタンスハンドルです。

コールバック関数はすべてキーワード_export でエクスポートしてください。たとえば，以下のようにコールバック関数を宣言します。

```
BOOL FAR PASCAL _export MyCallBack(HWND hWnd, DWORD lParam);
```

Windows のコールバック関数はすべて Pascal の呼び出しシーケンスをとるので，キーワード ***PASCAL*** 宣言します。

Windows のコールバック関数は ObjectWindows 環境外から呼び出され，C++対応の環境からは呼び出されません。C++の非スタティックメンバ関数をコールバック関数としては使えません。非スタティックメンバ関数は Windows にない隠された引数 this を利用するからです。しかし，スタティックメンバ関数は隠されたポインタ this を使わないのでコールバック関数として使えます。Windows にスタティックメンバ関数を呼び出させる場合は，必ずキーワード ***PASCAL*** 使ってスタティックメンバ関数を宣言してください。

列挙関数

Windows コールバック関数のうち最も頻繁に使われるのは，Windows 列挙関数の結果として呼び出されるものでしょう。関数を使うと，Windows システム内のある種類の要素を**列挙**することができます。たとえば，システムのフォントを列挙して，それぞれのフォントでサンプルテキストをプリントすることができます。列挙関数を使うには，前に述べた方法でコールバック関数を Windows に渡す必要があります。コールバック関数を必要とする Windows 関数には次のものがあります。

EnumChildWindows
EnumClipboardFormats
EnumFonts
EnumMetaFile
EnumObjects
EnumProps
EnumTaskWindows
EnumWindows

たとえば，以下のように宣言されたC++スタティックメンバ関数 *TMyWindow::ActOnWindow* があるとします。

```
class TMyWindow
{
:
static BOOL FAR PASCAL_export TMyWindow::ActOnWindow(HWND hWnd,
                                                      DWORD lParam);
:
};
```

これをコールバック関数として，Windows 関数 ***EnumWindows*** を呼び出すときに渡すことができます。

```
FARPROC lpEnumWinFunc;
:
lpEnumWinFunc = MakeProcInstance((FARPROC) TMyWindow::ActOnWindow,
                                 GetApplication()->hInstance);

ReturnVal = EnumWindow(lpEnumWinFunc, aLongVal);
FreeProcInstance(lpEnumWinFunc);
```

列挙関数のコールバックが 0 以外を返せば列挙は継続し，0 を返せば中止します。上の例のスタティックメンバ関数 *TMyWindow::ActOnWindow* は，渡されたハンドルが指定するウィンドウに対し何らかの処理を行ないます。引数 *aLongVal* は，***EnumWindows*** の呼び出し側がコールバック関数に渡す値です。

スマートコールバックの使用

「スマートコールバック」コンパイルオプションを使ってコードをコンパイルすると，***MakeProcInstance*** と ***FreeProcInstance*** を呼び出す必要がなくなります。このオプションは DS==SS となるようにプロローグコードとエピローグコードを変更します。つまり，デフォルトのデータセグメントをスタックセグメントと同じにします。前の例をスマートコールバックでコンパイルすると次のようになります。

```
ReturnVal = EnumWindow((FARPROC) TMyWindow::ActOnWindow, aLongVal);
```

Windows メッセージ

あるアプリケーションに影響を及ぼすイベントが発生したと判定すると，Windowsはそのアプリケーションにメッセージを送ります。メッセージは何種類もあり，その発生元も多様です。

- ユーザがマウスの移動やクリックあるいはキーボードのタイプをすると，ユーザイベントメッセージが生成されます。
- プログラムがプログラム自身に対して直接メッセージを送ることができます。
- プログラムからWindows関数を呼び出し，他のWindowsメッセージを受け取ることができます。
- ダイナミックデータエクスチェンジ（DDE）によってアプリケーションから他のアプリケーションにメッセージを送ることができます。
- Windows自体も各種のメッセージをアプリケーションに送ることができます。

Windowsアプリケーションは，送られてきたメッセージを取得し，これに応答する義務があります。特定ウィンドウに対する特定イベントが発生すると，Windowsアプリケーションがその取得・応答プロセスの一環として，応答を担当する関数を呼び出します。

ObjectWindowsでは，この取得と応答をObjectWindowsが代行してくれます。また送られてくるメッセージのデフォルト処理も請け負ってくれます。したがって，定義しなくてはならないのはアプリケーション固有の応答だけです。ObjectWindowsで提供されているクラスから独自のクラスを派生させ，そこで専用の**メッセージ応答メンバ関数**を宣言して定義してください。

以下は，マウスの左ボタンのクリックに応答するメンバ関数の宣言例です。これはObjectWindowsクラス ***TWindow*** からの派生クラス *TMyWindow* の定義の一部です。

```
class TMyWindow : public TWindow {
public:
:
  virtual void WMLButtonDown(RTMessage Msg) =
    [WM_FIRST + WB_LBUTTONDOWN];
:
};
```

上の宣言形式は，すべてのメッセージ応答メンバ関数に共通です。C++のメンバ関数宣言が拡張され，整数識別子と仮想メンバ関数が対応づけられるようになっています。この結果，ObjectWindowsではアプリケーションのメッセージを応答メンバ関数に割り当てることができます。

たとえば，*TMyWindow*のインスタンスがクリックされたときにビープ音を出させるには，以下のようにメッセージ応答メンバ関数を定義するだけで実現できます。

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0);        // Windows関数の呼び出し
```

Windowsメッセージの引数

メッセージの送り先として指定されたWindowsアプリケーションはメッセージを受け取るときに4つの引数を渡されます。そのうち，2つは対象となるウィンドウのハンドル(Windows識別子）とメッセージ識別子です。アプリケーションではこの2つの引数は必要ありません。ObjectWindowsが到着したメッセージをウィンドウオブジェクトの応答メンバ関数に割り当ててくれるからです。残りの2つの引数***WORD***値と***LONG***値に必要なデータが入っています。この2つの引数はObjectWindowsで定義済みの***TMessage***構造体にメンバ*WParam*とメンバ*LParam*として組み込まれており，その参照（***RTMessage***型）がメッセージ応答メンバ関数に渡されます。なお，***TMessage***構造体には，ObjectWindowsが内部的に設定するメンバ***Result***も含まれています。ただし，これは通常無視されます。

*WParam*は通常，ハンドル，識別子，コードなどの1つの情報を保持する***WORD***です。たとえば，メニュー選択によって***WM_COMMAND***メッセージが発行されますが，その*WParam*は選択されたメニュー項目のIDになります。

*LParam*はfarポインタなどの***LONG***値です。下位ワードと上位ワードを分けて使い，2つの情報を保持することもあります。たとえば，***WM_LBUTTONDOWN***（左ボタン押下）の応答メンバ関数に渡される*LParam*には，マウスがクリックされた位置のX座標とY座標が入ります。

上位 ***WORD*** と下位 ***WORD*** に個別アクセスできるように，*LParam* は ***WORD*** メンバとして *Lo* と *Hi* を持つ *LP* 構造体の共用体のメンバとして宣言されています。*Msg* を ***TMessage*** 構造体として宣言してあれば，*Msg.LP.Hi* と *Msg.LP.Lo* を使って *Msg.LParam* の上位ワードと下位ワードを参照できることになります。*TMessage* :: *WParam* も同様に *WP* 構造体の共用体のメンバとして宣言されています。ただし，このメンバ *Hi* と *Lo* は ***WORD*** ではなく ***BYTE*** です。

Windows メッセージの種類

Windows アプリケーションが受けるメッセージの種類を以下に示します。メッセージの一覧は **ObjectWindows for C++リファレンスガイド**の第5章『**Windows メッセージ**』にあります。

ウィンドウ管理メッセージ

ウィンドウの状態が変わると通知します。たとえば，ウィンドウがクローズすると，***WM_CLOSE*** が送出されます。また，ウィンドウの一部か全体の再表示が必要なときには ***WM_PAINT*** が送出されます。

初期化メッセージ

プログラムがメニューやダイアログボックスを作成するときに，***WM_CREATE*** や ***WM_INITDIALOG*** などが送出されます。

入力メッセージ

マウス，キーボード，スクロールバー，システムタイマによる入力があると受信されます。最も一般的な入力メッセージは ***WM_COMMAND*** です。このメッセージはメニューやアクセレータが選択されたとき，またはボタンのクリックのようなコントロールイベントが発生したときに発行されます。

システムメッセージ

コントロールメニューボックス，スクロールバー，最小化ボタン，最大化ボタンといったアプリケーションの標準アクセサリをユーザが操作したときに送出されます。たとえば，コントロールメニューボックスを選択すると ***WM_SYSCOMMAND*** が送出されます。ほとんどのアプリケーションは，システムメッセージを受け取りません。

クリップボードメッセージ

アプリケーションで所有または表示しているクリップボード内のデータに別のアプリケーションがアクセスしようとするとこのメッセージが送出されます。

システム情報メッセージ

色，フォントなど，Windows のシステムレベルの属性が変わると送出されます。

コントロール操作メッセージ

リストボックス，ボタンなどのコントロールに対してアプリケーションから送出されます。コントロールの属性を問い合わせたり設定したりするためのメッセージ群が，コントロールクラスごとに決まっています。コントロール操作メッセージは，MDI メッセージ以外のすべてのメッセージとは異なる点があります。すなわち，コントロールメッセージはアプリケーションにイベントを**通知**するものではなく，イベントを**発生**させるものなのです。後述の「**メッセージの送出**」を参照してください。

コントロール通知メッセージ

WM_COMMAND メッセージの一種で，リストボックス項目の選択，エディットコントロールへの入力などコントロールイベントの発生を親ウィンドウに通知します。メッセージの引数には，***LBN_SELCHANGE*** とか ***EN_CHANGE*** といった特定の通知コードが設定されます。

スクロールバー通知メッセージ

スクロールバーコントロール専用のコントロール通知メッセージです。***WM_HSCROLL*** と ***WM_VSCROLL*** の 2 つのメッセージがあります。

クライアント領域外メッセージ

WM_NCMOUSEMOVE や ***WM_NCPAINT*** などのメッセージで，入力メッセージに似ていますが，メニューバー，タイトルバー，境界枠など，ウィンドウのクライアント領域の外で発生したイベントだけに適用されます。普通，アプリケーションでこの種のメッセージのデフォルト処理を定義し直すことはありません。

マルチドキュメントインターフェース（MDI）メッセージ

MDI 標準方式に従うアプリケーションによって，子ウィンドウを操作するために送出されます。***WM_MDIACTIVATE*** と ***WM_MDIDESTROY*** というメッセージがあります。

デフォルトのメッセージ処理

Windows では，アプリケーションに送るメッセージの多くにデフォルト処理が用意されています。コントロール，ダイアログボックス，ウィンドウに対応するメッセージ応答メンバ関数の定義がなく，到着したメッセージが無視されると，ObjectWindows は Windows が提供するデフォルト処理を自動的に呼び出します。コントロール，ダイアログボックス，ウィンドウに対するこのようなデフォルト処理は，メンバ関数 ***DefWndProc*** で指定されます。

第 9 章『インターフェースオブジェクト』の「デフォルトメッセージ処理」を参照してください。

到着メッセージに対する応答メンバ関数を定義するとき，Windows が提供する応答も実行したい場合があります。このような場合は，メッセージ応答メンバ関数からコントロール，ダイアログ，またはウィンドウのメンバ関数 ***DefWndProc*** を呼び出してください。

メッセージの送出

アプリケーションプログラム自体もメッセージを発行できます。***SendMessage*** と ***PostMessage*** という2つのWindows API関数を呼び出すと，アプリケーションプログラム自体にメッセージを送ることができます。***SendMessage*** はハンドルで指定したウィンドウにメッセージの即時処理を強制し，***PostMessage*** はこのウィンドウのアプリケーションキューにメッセージを登録し，その処理を待たずに終了します。この方法は，アプリケーションプログラム内部からイベントをシミュレートする場合にも便利です。API関数の詳細については，オンラインヘルプ，または**ObjectWindows for C++リファレンスガイド**の第4章『**Windows関数**』を参照してください。

リストボックスやボタンのようなコントロールを操作する場合には，***SendMessage*** が最も便利です。Windowsでは，リストボックスへの項目の追加（***LB_ADDSTRING***）とか，チェックボックスの状態変更（***BM_SETCHECK***）といった処理を行なうための**コントロール操作メッセージ**が定義されています。また，ObjectWindowsのコントロールオブジェクトには，***TListBox*** : : ***AddString*** や ***TCheckBox*** : : ***SetCheck*** というメンバ関数が定義されています。これらのメンバ関数によって，上に述べた共通的なメッセージが該当するインターフェース要素に送出されます。もっとあいまいなメッセージをウィンドウのコントロールに送出したければ，コントロールの *HWindow* を第1引数（*Wnd*）として渡す ***SendMessage*** を使ってください。

177ページの「ダイアログコントロールの操作」を参照してください。

ダイアログボックスのコントロールにメッセージを送る場合の手順は少々異なります。多くの場合，***TDialog*** オブジェクトのコントロールには，対応するObjectWindowsインターフェースオブジェクトがありませんが，***TDialog*** : : ***SendDlgItemMsg*** を使えばメッセージを送ることができます。すなわち，***TWindow*** のコンストラクタをかわりに使ってインターフェースオブジェクトを対応づけることによって，ダイアログボックスのコントロールと通信ができるようになります。

メッセージ範囲

メッセージは ***WORD*** 型の値で定義されるため，最大 65, 536 個のメッセージが利用可能です。ObjectWindows では，標準の Windows メッセージ，コマンドメッセージ，通知メッセージというように，それぞれ特定の範囲にメッセージを区分しています。範囲を識別しやすくするために，ObjectWindows ではそれぞれの範囲を区切るオフセットとして定数が定義されています。メッセージの範囲と ObjectWindows の定数を**表 7.1** に示します。

表 7.1　Windows メッセージの範囲

定数	値	範囲	意味
WM_FIRST	0x0000	0x0000-0x3FFF	Windows メッセージ
WM_USER	0x0400	0x0400-0x7F00	プログラマ定義のウィンドウメッセージ
WM_INTERNAL	0x7F00	0x7F00-0x7FFF	内部用に予約
ID_FIRST	0x8000	0x8000-0x8EFF	プログラマ定義の子 ID メッセージ
ID_INTERNAL	0x8F00	0x8F00-0x8FFF	内部用に予約
NF_FIRST	0x9000	0x9000-0x9EFF	プログラマ定義の通知メッセージ
NF_INTERNAL	0x9F00	0x9F00-0x9FFF	内部用に予約
CM_FIRST	0xA000	0xA000-0xFEFF	プログラマ定義のコマンドメッセージ
CM_INTERNAL	0xFF00	0xFF00-0xFFFF	内部用に予約

ユーザ定義メッセージ

Windows では，アプリケーションで使用する独自のメッセージを定義できます。ユーザ定義メッセージは，新しいイベントを定義し，これに応答する場合に便利です。たとえば，1 つのウィンドウからアプリケーションでオープンしている他のすべてのウィンドウにユーザ定義メッセージを送ることができます。

それぞれの Windows メッセージには，***WORD*** 型の識別子がついています。定義済みの各 Windows メッセージに対しては，WINDOWS.H に ***WORD*** 型の定数が定義されています。この定数は，メッセージ応答メンバ関数の宣言に使われているものです。

Windowsメッセージ識別子の有効範囲内には，ユーザ定義メッセージに使う値が予約されています。定義済みのWindowsメッセージに予約されている値の範囲は0から ***WM_USER*** −1です。また，ユーザ定義メッセージの値の範囲は ***WM_USER*** から ***WM_USER*** +31,744です。ユーザメッセージの値を定数として定義する場合，***WM_USER***，***WM_USER*** +1，***WM_USER*** +2のように始めることをお勧めします。次に例を示します。

```
#define WM_MYFIRSTMESSAGE   WM_USER
#define WM_MYSECONDMESSAGE WM_USER + 1
#define WM_MYTHIRDMESSAGE   WM_USER + 2
```

Windows関数の ***SendMessage*** を使ってユーザメッセージを送出するには次のようにします。

```
SendMessage(AWindow->HWindow, MW_MYFIRSTMESSAGE, AWord, ALong);
```

ユーザ定義メッセージを受信するには，標準のWindowsメッセージの場合と同じ手順に従います。一例を示します。

```
class  MyWindow : public TWindow
{
:
 virtual void WMMyFirstMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYFIRSTMESSAGE];
  virtual void WMMySecondMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYSECONDMESSAGE];
  virtual void WMMyThirdMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYTHIRDMESSAGE];
:
};
```

第8章

モジュールとアプリケーションオブジェクト

ObjectWindowsアプリケーションではまず最初に，ObjectWindowsの***TApplication***クラスからアプリケーションクラスを派生させて定義しなければなりません。アプリケーションオブジェクトは，ObjectWindowsアプリケーションから以下の動作を継承します。

- アプリケーションのメインウィンドウを作成して表示する。
- 通信ポートの構成などアプリケーションの最初のインスタンスを，アプリケーションの全インスタンスに共通なタスク用に初期化する。
- アプリケーションの各インスタンスをアクセラレータテーブルのロード用などに初期化する（同じObjectWindowsアプリケーションの複数のインスタンスを同時に実行できる）。
- アプリケーションで受信するWindowsメッセージを処理する。
- アプリケーションをクローズする。

TApplicationからの派生クラスを定義するだけでなく，メインウィンドウオブジェクトを構築する機能も追加する必要があります。また，インスタンスの初期化，アプリケーションのクローズ，Windowsメッセージの処理用として規定されているデフォルトの動作を定義し直してもかまいません。

アプリケーションの流れ

ObjectWindowsアプリケーションのメインプログラムは，通常3つの文だけで構成されます。

THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance, lpCmdLine, nCmdShow)；HelloApp.Run ()；

最初の文でコンストラクタを呼び出してアプリケーションオブジェクトを構築します。このコンストラクタはアプリケーションオブジェクトのデータメンバの初期化も行ないます。***GetApplication*** を呼び出せば，ObjectWindows プログラムの他の部分からでもこのアプリケーションオブジェクトにアクセスできます。

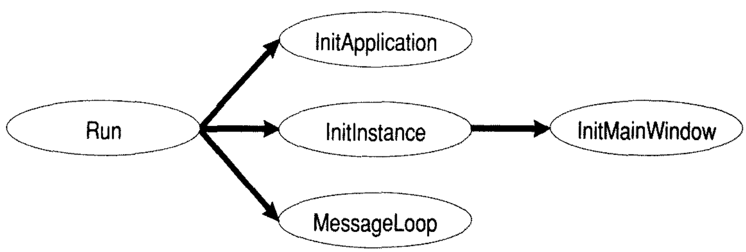
2番目の文でアプリケーションのメンバ関数 ***Run*** を呼び出します。このメンバ関数が ***InitApplication*** と ***InitInstance*** を呼び出して最初のインスタンスと以降の個々のインスタンスを初期化させます。続いて ***InitMainWindow*** が呼び出されて，メインウィンドウを作成します。たいていの場合は，***InitMainWindow*** だけを定義し直せばいいはずです（図8.1参照)。

次に，***Run*** が ***MesssageLoop*** を呼び出してアプリケーションを起動し，受信した Windows メッセージの処理を開始させます。このメッセージによってアプリケーションの処理が指示されます。***MessageLoop*** は受信メッセージに該当するメンバ関数を呼び出して受信メッセージを処理させます。***MessageLoop*** とは，その名前のとおり，アプリケーションがクローズするまで実行し続けるメッセージループのことです。

return HelloApp.Status；

3番目の文で ObjectWindows がデータメンバ ***Status*** に格納しているアプリケーションの最終状態を返します。これがないと，***WinMain*** は int を返せません。***Status*** は0以外の値でエラーを示すので，デバッグにも使えます。

図8.1
アプリケーションの流れを制御するメンバ関数呼び出し



アプリケーションの初期化

一連のメンバ関数呼び出しによってアプリケーションオブジェクトが初期化されるので，メンバ関数を定義し直すだけで，処理の多くの部分をカスタマイズすることができます。新たに記述が必要なメンバ関数は，***InitMainWindow*** だけです。***InitInstance*** を定義し直せば，ObjectWindows アプリケーションで実行中の各インスタンスに応じた初期化ができます。また，***InitApplication*** を

定義し直せば，最初に実行されるインスタンスを初期化できます。

メインウィンドウの初期化

次に示すのは，アプリケーションオブジェクトクラスに最低限必要な定義です。

このコードはHELLO-APP.CPPの一部です。

```
class THelloApp :public TApplication
{
public:
    THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
              LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
              hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
    virtual void InitMainWindow();
};
```

メンバ関数 ***InitMainWindow*** は必ず定義しておきます。これによって，メインウィンドウオブジェクトが構築され，アプリケーションオブジェクトのデータメンバ ***MainWindow*** に格納されます。メンバ関数 ***InitMainWindow*** の簡単な例を次に示します。

```
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}
```

この単純なObjectWindowsアプリケーションはメインプログラムと ***TApplication*** の派生クラスの定義で構成されます。この派生クラスではメンバ関数 ***InitMainWindow*** だけが定義されます。

これはHELLOAPP.CPPです。

```
#include <owl.h>

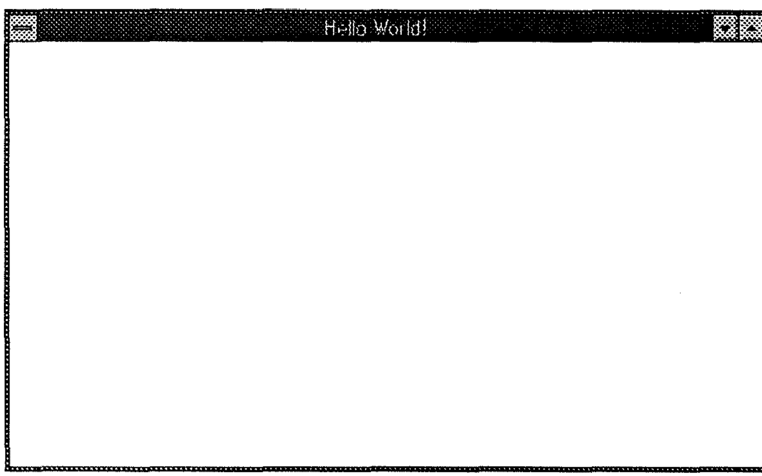
// TApplicationからの派生クラスを定義
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
            hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

```
//  THelloAppのMainWindowデータメンバを構築
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance,
                      lpCmdLine, nCmdShow);
  HelloApp.Run();
  return HelloApp.Status;
}
```

図8.2に示すように，このプログラムは，“Hello World！”というタイトルのついたウィンドウを表示するだけの最小構成のアプリケーションです。このウィンドウは移動とサイズ変更が可能です。また，右上隅の↓アイコンをクリックすることによって，このウィンドウをアイコンに最小化することもできます。これを元に戻すには，アイコンをダブルクリックします。↑アイコンをクリックすると最大化され，画面いっぱいに表示されます。ウィンドウをクローズし，アプリケーションを終了するには，左上隅のコントロールメニューボックスをダブルクリックします。

図8.2
The Main Window



各アプリケーションインスタンスの初期化

アプリケーションクラスのメンバ関数 ***InitInstance*** は，Windows アプリケーションの各インスタンスを初期化します。このメンバ関数は ***TApplication*** で定義されており，***InitMainWindow*** を呼び出し，さらにメンバ関数 ***Show*** を呼び出してメインウィンドウを表示します。このため，***InitInstance*** を定義し直すだけで，標準の初期化処理を変更できます。応用例としては，アプリケーション固有の処理を規定したアクセラレータテーブルをロードする場合が挙げられます。なお，アプリケーションに合わせて ***InitInstance*** を定義し直す場合には，必ず ***TApplication*** :: ***InitInstance*** を最初に呼び出すようにします。

以下にメンバ関数 ***InitInstance*** の例を示します。この例は，アプリケーションが実行される前にアクセラレータテーブルをロードするものです。100 というのはアクセラレータテーブルのリソース識別子で，リソースファイルに定義されています。

```
void TMyApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(GetApplication()->hInstance,
                               MAKEINTRESOURCE(100));
}
```

InitInstance はアプリケーションインスタンスの初期化に限ります。メインウィンドウは，コンストラクタとメインウィンドウオブジェクトの ***SetupWindow*** メンバ関数で初期化します。

最初のアプリケーションインスタンスの初期化

アプリケーションを終了せず同時に何回も実行する場合，最初に行なわれる処理を定義することができます。この処理を第 1 インスタンスの初期化と呼ぶことにします。なお，アプリケーションを起動して終了し，また起動するというような場合，それぞれのインスタンスが第 1 インスタンスとみなされます。

現在のインスタンスが第1インスタンスである場合，所定のメンバ関数 ***InitApplication*** が呼び出されます。「無動作（do-nothing）」のメンバ関数 ***InitApplication*** が ***TApplication*** に定義されており，これを定義し直すことによって，第1インスタンス専用の初期化を行なうことができます。

たとえば，*TestApp* を変更して第1インスタンスであるかどうかをメインウィンドウのタイトルに表示させる場合，データメンバ *WindowTitle* をアプリケーションクラスに追加します。次に，*WindowTitle* に“First Instance”を設定するようメンバ関数 ***InitApplication*** を定義します。さらにこのアプリケーションコンストラクタに *WindowTitle* を初期化させ，“Additional Instance”を設定します。***InitApplication*** は，最初のインスタンスのときだけに呼び出されます。

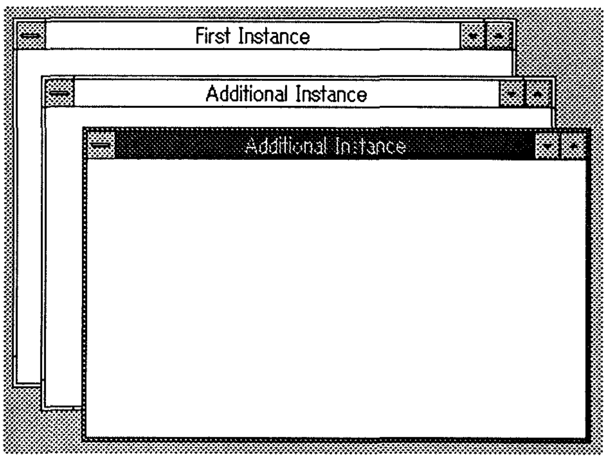


図 8.3
改良したアプリケーションの初期化表示

これは INSTTEST .CPP です。

```
#include <owl.h>
#include <string.h>

class TTestApp : public TApplication
{
public:
  TTestApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
           lpCmdLine, int nCmdShow);

protected:
  char WindowTitle[20];
  virtual void InitMainWindow();
  virtual void InitApplication();
};

TTestApp::TTestApp(LPSTR AName, HANDLE hInstance, HANDLE
                   hPrevInstance, LPSTR lpCmdLine, int nCmdShow) :
                   TApplication(AName, hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow)
{
  strcpy(WindowTitle, "Additional Instance");
}

void TTestApp::InitApplication()
{
  strcpy(WindowTitle, "First Instance");
}

void TTestApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, WindowTitle);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TTestApp TestApp("Instance Tester", hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

アプリケーションの実行

プログラムでアプリケーションオブジェクトのメンバ関数 ***Run*** を呼び出すと，メンバ関数 ***MessageLoop*** が呼び出されるので，アプリケーションのメッセージループが起動されます。プログラムの動作が行なわれている間，受信されたWindowsメッセージはメッセージループによって処理されます。ObjectWindowsプログラムは，このように自動的に実行するメンバ関数 ***MessageLoop*** を継承します。

メンバ関数 ***MessageLoop*** は，3つの翻訳メンバ関数を呼び出してWindowsメッセージを処理させます。このうち，***ProcessDlgMsg*** はモードなしダイアログメッセージを処理します。***ProcessAccels*** はアクセラレータを処理します。***ProcessMDIAccels*** はMDIアプリケーションのアクセラレータを処理します。アクセラレータやモードなしダイアログボックスを使わないアプリケーション，あるいはMDIアプリケーション以外のアプリケーションの場合，***MessageLoop*** はもっと簡単なものになります。翻訳メンバ関数の指定項目については，**ObjectWindows for C++リファレンスガイド**の第1章**『ObjectWindows クラス』**にある ***TApplication*** を参照してください。

アプリケーションのクローズ

アプリケーションの終了前に，プログラムはメッセージループから抜け出さなければなりません。抜け出しは，アプリケーションのメインウィンドウをユーザがクローズしようとしたときに行なわれる場合もあります。わざわざ「場合もある」というのは，ObjectWindowsではアプリケーションのクローズ方法を定義し直すことができるからです。たとえば，クローズ前にファイルがセーブされているかどうかをチェックさせることもできます。

コントロールメニューボックスをダブルクリックしてアプリケーションをクローズしようとすると，次の処理が行なわれます。

1. Windowsは，*WM_CLOSE* メッセージをアプリケーションのメインウィンドウに送ります。
2. メインウィンドウオブジェクトはこのメッセージを受け取ると，メンバ関数 ***CloseWindow*** を呼び出します。この ***CloseWindow*** は，***ShutDownWindow*** を呼び出し，メインウィンドウをクローズする前に，必ずメンバ関数 ***CanClose*** を呼び出してアプリケーションオブジェクトをチェックします。アプリケーションのメインウィンドウをクローズすると，アプリケーション自体をクローズすることになるからです。
3. アプリケーションオブジェクトのメンバ関数 ***CanClose*** は，メインウィンドウからのメンバ関数 ***CanClose*** の呼び出しの結果を返します。これで，メインウィンドウ自体でクローズすべきかどうかを決められます。
4. メインウィンドウはデフォルトで，子ウィンドウにクローズしてもよいかどうかを問い合わせ，メインウィンドウ自体をクローズしてよいかどうかを判断します。すべての子ウィンドウのメンバ関数 ***CanClose*** が真を返すと，メインウィンドウの ***CanClose*** も真を返し，アプリケーションがクローズします。

この方法は，「アプリケーションをクローズしてはならない理由を誰か知っていたら教えて欲しい」と告げることと同じです。最終的には，アプリケーションをクローズするための承認を与えるかどうかは，アプリケーションオブジェクトにまかされます。アプリケーションオブジェクトはデフォルトで，この承認を与える前にメインウィンドウをチェックすることになっています。メインウィンドウクラスのメンバ関数 ***CanClose*** を定義し直して別のクローズ方法をとることもできます。アプリケーションクラスのメンバ関数 ***CanClose*** を定義し直して，デフォルトのクローズ方法を変えることもできます。

第9章

インターフェースオブジェクト

ウィンドウ，ダイアログボックス，コントロールを表わすオブジェクトのことをユーザインターフェースオブジェクト，または単にインターフェースオブジェクトといいます。この章では，インターフェースオブジェクトの一般的な必須条件や動作，また画面に表示される実際のウィンドウ，ダイアログボックス，コントロールとの関係について説明します。

また，Windows メッセージを交換する仕組みに加えて，アプリケーションで使用される様々なインターフェースオブジェクト相互間の関係についても解説することにします。

TWindowsObject

TWindowsObject は，Windows インターフェースクラスである ***TDialog***，***TWindow***，および ***TWindow*** からの派生クラス ***TControl*** に共通な基本クラスであり，ウィンドウ，ダイアログ，コントロールの各オブジェクトに共通した動作を定義しています。***TWindowsObject*** のメンバ関数には次の機能があります。

- ObjectWindows オブジェクトと Windows 要素という両面構造をもち，オブジェクトの構築と破棄，および要素の作成と廃棄を行なう。
- インターフェースオブジェクト間で自動的に親子関係を実現する（クラスの継承関係ではない）。
- 新しい Windows クラスを登録する（第10章『**ウィンドウオブジェクト**』参照）。

TWindowsObject の働きは意識されません。たとえ意識されたとしても，***TWindowsObject*** から直接新しいクラスを派生させることはまれです。***TWindow*** と ***TDialog*** から新しいクラスを派生させたときにオブジェクトが継承する機能の多くが ***TWindowsObject*** に定義されています。

インターフェースオブジェクトが必要な理由

Windows にはすでに視覚的なウィンドウやダイアログ，コントロールがあるのに，どうしてインターフェースオブジェクトが必要なのでしょうか。

それぞれのインターフェースオブジェクトには，対応する**ビジュアルインターフェース要素**があります。これはオブジェクトではなく，物理的なウィンドウ，ダイアログ，コントロールのことです。このようなビジュアルインターフェース要素に対して，インターフェースオブジェクトがオブジェクト指向の機能を代行するのです。インターフェースオブジェクトは，対応するインターフェース要素の作成，初期化，管理，廃棄を行なうメンバ関数を提供します。また，インターフェースオブジェクトのデータメンバはインターフェース要素のハンドル，親ウィンドウや子ウィンドウなどの関連データを保持します。インターフェースオブジェクトのメンバ関数は，Windows プログラミングに必要な詳細部分の多くをプログラマのかわりに処理してくれるものです。

オブジェクトと要素の関係は，C＋＋ストリームと DOS ファイルの関係に似ています。DOS ファイル I/O の詳細を処理させるのにストリームオブジェクトを構築するように，ウィンドウ I/O の詳細を処理させるのにウィンドウオブジェクトを構築します。

ウィンドウ，ダイアログ，コントロールを適切に処理するためには，インターフェースオブジェクトと対応するインターフェース要素という構造をプログラマの側で意識する必要があります。たとえば，完全なインターフェースオブジェクトを作成する場合には，**2つ**のメンバ関数を呼び出さなければなりません。最初のメンバ関数はコンストラクタです。これによってインターフェースオブジェクトのインスタンスを構築し，スタイルやメニューがある場合にはこの属性を設定します。

2つ目のメンバ関数は ***MakeWindow*** という作成メンバ関数です。インターフェースオブジェクトと新しいインターフェース要素がこのメンバ関数で対応づけられます。この対応づけは，インターフェースオブジェクトのデータメンバ

HWindow で管理します。ウィンドウを指すハンドルは，このデータメンバに保持されています。また，***MakeWindow*** はアプリケーションオブジェクトのメンバ関数になっており，オブジェクトのメンバ関数 ***Create*** を呼び出します。こうして，実際にビジュアル要素を作成するよう Windows に指示が出されるのです。

MakeWindow は「安全に」ビジュアル要素を作成します。***Create*** とは異なり，***MakeWindow*** はインターフェースオブジェクトを確実に構築し，充分なメモリを確保します。多少のオーバーヘッド時間を要しますが，メモリ不足になって Windows とのクラッシュを起こす心配がなくなります。

同様に，ビジュアルインターフェース要素を**破棄**し，対応するインターフェースオブジェクトを**解放**しなければなりません。***ShutDownWindow*** を呼び出せば，インターフェース要素を破棄し，インターフェースオブジェクトを削除してくれます。

なお，インターフェースオブジェクトとこれに対応するビジュアル要素の両方を作成したからといって，必ずしも何かが画面に表示されるというわけではありません。ビジュアル要素が作成されると，ウィンドウにスタイル ***WS_VISIBLE*** がセットされているかどうかを Windows でチェックします。***WS_VISIBLE*** などのウィンドウスタイルのセットまたはクリアは，インターフェースオブジェクトのデータメンバ *Attr.Style* の設定により，コンストラクタで行ないます。***WS_VISIBLE*** がセットされている場合のみ，インターフェース要素は表示されます。

いずれの時点でも，インターフェースオブジェクトのメンバ関数 ***Show*** を呼び出すことによって，要素の表示，非表示を行なえます。

ウィンドウの親子関係

Windows アプリケーションでは，ウィンドウ，ダイアログボックス，コントロールといったインターフェース要素は，親と子の結合によって関係づけられます。一方が親ウィンドウになり，他方が子ウィンドウになっているときは，2つのインターフェース要素が関係しています。この親子関係は，オブジェクト関係を意味する継承とかインスタンスの所有関係とは異なります。子ウィンドウを必ず親ウィンドウから派生させたり，何かを継承させる必要はありません。

子ウィンドウは，別のインターフェース要素によって管理されるインターフェース要素です。たとえば，リストボックスは表示先のウィンドウやダイアログボックスで管理されます。リストボックスが表示されるのは，対応する親ウィンドウが表示されるときだけです。一方，ダイアログボックスは生成元のウィンドウによって管理されます。親ウィンドウをクローズすると，子ウィンドウも自動的にクローズし，親ウィンドウを移動すると，子ウィンドウも一緒に移動します。

インターフェース要素であれば，ウィンドウ，ダイアログ，コントロールを問わず，親ウィンドウまたは子ウィンドウになれます。

子ウィンドウリスト

インターフェース要素の親は，必ずインターフェース要素の構築時に指定します。親ウィンドウオブジェクトは，インターフェースオブジェクトのコンストラクタの引数にします。子ウィンドウオブジェクトは，親ウィンドウオブジェクトを指すポインタをデータメンバ ***Parent*** に格納することによって，親ウィンドウオブジェクトとの関係を保持します。また，下位の子ウィンドウオブジェクトとの関係を自動的に保持します。

子ウィンドウを持つ新しいインターフェースクラスを定義する場合には，子ウィンドウオブジェクトも構築するようコンストラクタを定義します。そうすれば，アプリケーションで親ウィンドウのメンバ関数 ***Create*** を呼び出すと，親のインターフェース要素が作成されます。正常に作成された場合には，作成の過程でメンバ関数 ***SetupWindow*** が呼び出されます。***TWindowsObject*** : : ***SetupWindow*** は非公開の子リストに載っているすべての子ウィンドウに対してメンバ関数 ***Create*** を呼び出します。ただし，デフォルトでダイアログはこの処理の対象になりません。***WS_VISIBLE*** を設定されている子ウィンドウは作成と同時に表示されます。

子ウィンドウを作成した後でなんらかの設定作業を行なわせるために，親ウィンドウの ***SetupWindow*** を定義し直すこともよくあるはずです。たとえば，リストボックスに要素を書き込む場合などです。このような場合には，定義し直したメンバ関数 ***SetupWindow*** の第 1 行目で，ウィンドウオブジェクトの基本クラスのメンバ関数 ***SetupWindow*** を呼び出してください。

SetupWindow を定義し直す例については，87 ページのステップ 10 を参照してください。

ポップアップウィンドウなどの子ウィンドウをこのデフォルト処理から明示的に除外するには，オブジェクトを構築した後でメンバ関数 ***DisableAutoCreate*** を呼び出します。また，ダイアログボックスなど，通常は対象外の子ウィンドウ

に対してデフォルト処理を明示的に要求するには，メンバ関数 ***EnableAutoCreate*** を呼び出します。

親ウィンドウのメンバ関数 ***Create*** を呼び出すと，その子ウィンドウのメンバ関数 ***Create*** が呼び出されるように，親のデストラクタを呼び出すと，その子ウィンドウのデストラクタが呼び出されます。メンバ関数 ***CanClose*** は，非公開リスト内の子ウィンドウオブジェクトに対し，各子ウィンドウオブジェクトのメンバ関数 ***CanClose*** を呼び出します。この場合，すべての子ウィンドウが真を返すと ***CanClose*** も真を返します。

子ウィンドウの繰り返し処理用メンバ関数

ウィンドウの各子ウィンドウに対し繰り返して処理を実行するメンバ関数を記述する場合があります。たとえば，ウィンドウ内のすべての子チェックボックスをチェックする場合などです。このような場合，以下の例のように ***TWindowsObject*** から継承したメンバ関数 ***ForEach*** を使います。

ForEach と ***FirstThat*** の両メンバ関数は，第１引数として関数へのポインタをとります。***TWindowsObject*** : : ***ForEach*** と ***TWindowsObject*** : : ***FirstThat*** については，**ObjectWindows for C＋＋リファレンスガイド**を参照してください。

```
void CheckTheBox(Pvoid P, Pvoid Param)
{
  (PTCheckBox)P->Check();
}

void TMyWindow::CheckAllBoxes();
{
  ForEach(CheckTheBox, NULL);
}
```

また，特定の子ウィンドウを見つけるために，ウィンドウ内の一連の子ウィンドウに対し繰り返し処理を行なうメンバ関数を記述する場合もあります。たとえば，多くのチェックボックス子ウィンドウがあるウィンドウ内で，チェックされている最初のチェックボックスを見つける場合などです。こういった場合には，次のように ***TWindowsObject*** から継承したメンバ関数 ***FirstThat*** を使います。

```
BOOL IsThisBoxChecked(Pvoid P, Pvoid Param)
{
  return ((PTCheckBox)P->GetCheck() == BF_CHECKED);
}

PTCheckBox TMyWindow::GetFirstChecked();
{
  return (FirstThat(IsThisBoxChecked, NULL))
}
```

メッセージ処理

ObjectWindowsプログラムとWindowsの間では2方向の通信が行なわれます。1つの方向では，アプリケーションがWindows関数を呼び出して，ユーザインターフェースを制御します。もう1つの方向では，プログラムの**イベント**に応じてWindowsがアプリケーションにメッセージを送ります。メニュー項目を選択したときの***WM_COMMAND***や，マウスの左ボタンをクリックしたときの***WM_LBUTTONDOWN***などが一例です。100以上もの標準Windowsメッセージが用意されていますが，**ユーザ定義**のメッセージを作成することもできます。これに関しては第7章**『ObjectWindowsの概要』**に説明があります。

Windowsアプリケーションは，Windowsから受信したメッセージに応じて処理を行ないます。受信したWindowsメッセージに別々に応答することで，アプリケーション固有な動作を実現するわけです。メッセージ処理中にWindows関数が呼び出されることもあり，Windows関数自体が新しいWindowsメッセージを生成することもあります。

受信メッセージは，必ず特定のウィンドウ，すなわち現在のイベントが最も影響を及ぼすウィンドウあてになります。たとえば，あるウィンドウ内でマウスの左ボタンをクリックしたときに発生する***WM_LBUTTONDOWN***メッセージは，そのウィンドウを対象とします。ObjectWindowsでは，ウィンドウへのメッセージがウィンドウの応答メンバ関数に対応づけられます。

このWindowsメッセージとオブジェクトメンバ関数の連係動作が，ObjectWindowsアプリケーションの骨組みになっています。ある意味で，ObjectWindowsアプリケーションの動作の中心は，Windowsイベントに応答することにあるともいえます。これまでのプログラム主動型DOSアプリケーションの方向とはまったく異なる，メッセージに基づいたイベント駆動型のプログラミングです。イベント駆動型プログラムは，**話しかけられないと何も話さない**

人にたとえることができるでしょう。そのため，いろいろと教えてやることが多いかもしれませんが，ユーザインターフェースがよくなり，マルチタスク可能なアプリケーションになるという利点を考えれば補って余りあるものがあります。

メッセージへの応答

メッセージ応答メンバ関数の詳細については18ページを参照してください。

受信した Windows メッセージに応答するようウィンドウオブジェクトを設計する際，処理を必要とする受信メッセージのそれぞれに1対1で対応するメンバ関数を記述します。このようなメンバ関数のことを**メッセージ応答メンバ関数**といいます。この特別なメッセージとメンバ関数の対応は，仮想関数宣言を拡張することによって実現しています。メンバ関数をメッセージ応答メンバ関数とするには，***WM_FIRST*** とメッセージ名定数を合わせたものを，次のようにして仮想メンバ関数宣言の最後に追加します。

```
class TMyWindow : public TWindow {

public:
:
  virtual void WMRButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_RBUTTONDOWN];
:
};
```

上の例では，*TMyWindow* オブジェクトが ***WM_RBUTTONDOWN*** メッセージを受け取ると，メンバ関数 ***WMRButtonDown*** が呼び出されます。メンバ関数名をメッセージ名に似せる必要はありませんが，似ているとプログラムが読みやすくなります。ObjectWindows はこの方法をインターフェースクラスに採用しています。

Msg は ***RTMessage*** 型の必須引数です。***RTMessage*** は ObjectWindows の定義済みの型で，***TMessage*** 構造体への参照を表わします。***TMessage*** には *WParam*, *LParam*, *Result* の各メンバがあります。***WORD*** 型のメンバ *WParam* と ***LONG*** 型のメンバ *LParam* には，メッセージを発生させたイベントに関する情報が入ります。1つの引数に2つの情報を保持する場合もあります。たとえば，***WM_LBUTTONDOWN*** が送られると，*LParam* の下位 ***WORD*** でクリック位置の X 座標を保持し，上位 ***WORD*** で Y 座標を保持します。

LParam は *LP* 構造体の共用体のメンバとして宣言されます。*LP* 構造体は ***WORD*** 型の *Hi* メンバと *Lo* メンバを宣言します。したがって *Msg.LP.Hi* と *Msg.LP.Lo* に分けて使い，*Msg.LParam* の上位ワードと下位ワードを参照できます。同様に，*WParam* は *WP* 構造体の共用体のメンバとして宣言されます。よく使われるのは *WParam*，*LP.Lo*，*LP.Hi* の3つの ***WORD*** 型メンバです。*Msg* は参照で渡されます。この理由は，Windows メッセージのなかに応答を必要とするものがいくつかあり，この応答をメンバ関数で *Result* メンバに格納しなければならないからです。

ObjectWindows プログラムでは，オブジェクトが応答しない Windows メッセージを無視することができます。ここで無視というのは，メッセージに応答すべきメンバ関数を定義しないでおくことです。このようなメッセージはデフォルトで，ウィンドウが ***TWindowObject*** から継承したメンバ関数 ***DefWndProc*** によって処理されます。

メニューおよびエディットコントロールやスクロールバーなどの子ウィンドウに対してよく使われるメッセージでは，ObjectWindows のメッセージ応答モデルのかわりになるショートカットを使うことができます。このショートカットについては以降の節で説明します。

コマンドメッセージと子ウィンドウメッセージ

ユーザがメニュー項目を選択したりアクセラレータをタイプすると**コマンドメッセージ**が生成されます。また，ボタンやリストボックスなどのコントロールをアクティブにすると**子 ID メッセージ**が生成されます。たいていの場合，結果として ***WM_COMMAND*** メッセージが発行されます。

ObjectWindows では，あらゆる可能性に対処し，かつ長い case 文を避けるための工夫が施されています。すなわち，メッセージの内容に基づいて別のもっと専用化された仮想メンバ関数を呼び出し，自動的に ***WM_COMMAND*** メッセージに応答します。応答するには，すでに記述したのと同様のメンバ関数宣言の拡張構文を使って，コマンドメッセージと子 ID メッセージに対する応答メンバ関数を定義します。

コマンドメッセージ処理

メニューやアクセラレータが生成したイベントとコマンドメッセージ応答メンバ関数とは識別子を使って対応づけます。この識別子としてObjectWindowsで定義されている定数 ***CM_FIRST*** とメニューリソースまたはアクセラレータリソースで定義されているメニューIDまたはアクセラレータIDの和を使います。たとえば，***TFileWindow*** のオブジェクトであるウィンドウにFile | New，File | Open，File | Saveという3つのメニュー項目がある場合を考えてみましょう。OWLRC.Hに定義されているメニューIDは，それぞれ ***CM_FILENEW***，***CM_FILEOPEN***，***CM_FILESAVE*** です。この場合，***TFileWindow*** オブジェクトの定義は次のようになります。

```
class TFileWindow : public TWindow
{
  :
  virtual void CMFileNew(RTMessage Msg) =
    [CM_FIRST + CM_FILENEW];
  virtual void CMFileOpen(RTMessage Msg) =
    [CM_FIRST + CM_FILEOPEN];
  virtual void CMFileSave(RTMessage Msg) =
    [CM_FIRST + CM_FILESAVE];
  :
}
```

ただし，メニューを持つウィンドウに，必ずしもコマンドメッセージに対する応答を定義しておく必要はありません。コントロールなどの子ウィンドウでかわりに応答を定義できるからです。たとえば，メニューやアクセラレータが選択されたときに入力フォーカスを受け取る子ウィンドウに応答メンバ関数を定義しておけば，この子ウィンドウでまずコマンドメッセージに応答することができます。子ウィンドウに応答が定義されていなければ，メッセージはその子ウィンドウの親ウィンドウに送られ，さらにその親ウィンドウに送られるというように，応答が得られるまで，またはメニューのオーナーウィンドウに達するまでこの処理が続きます。すなわち，ウィンドウ内のエディットコントロール自身が，エディットメニューの選択に直接応答することができるのです。この手法の例についてはObjectWindowsの ***TEditWindow*** クラスを参照してください。

子メッセージの処理

ユーザがボタンのクリック，エディットコントロールへのタイプなど，コントロール（子ウィンドウ）に影響を及ぼすような操作を行なうと，その親ウィンドウオブジェクトまたはダイアログオブジェクトは子ウィンドウから**通知メッセージ**を受信します。親ウィンドウを通知メッセージに応答させるには，識別子のついたメンバ関数を定義しなければなりません。この識別子として，ObjectWindowsで定義されている定数 ***ID_FIRST*** と子ウィンドウIDの和を使います。子ウィンドウIDはダイアログリソースに定義されているもの，またはコントロールオブジェクトの生成時に定義されたものです。たとえば，リストボックスを持つウィンドウの場合は ***HandleListBoxMsg*** というメンバ関数を定義して，以下のようにリストボックスからの子IDメッセージに応答させます。

```
class TListBoxWindow : public TWindow
{
  :
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
}
```

ここで，***ID_LISTBOX*** は，コントロールIDに相当するものとして定義されている定数です。子ウィンドウIDは4,096未満の正の整数にします。

親ウィンドウで子イベントメッセージに応答するかわりに，コントロールに**通知別の**応答メンバ関数を定義することもできます。第12章『**コントロールオブジェクト**』の節「**コントロール通知メッセージに対する応答**」を参照してください。

デフォルトのメッセージ処理

通知メッセージについては，第12章『コントロールオブジェクト』を参照してください。

この節の説明は，***WM_FIRST*** 定数によって直接受け取るWindowsメッセージを対象にしたものです。コマンドメッセージ，通知メッセージ，子IDメッセージは対象外です。

Windowsアプリケーションは，アプリケーションで応答しなくてもよいメッセージを多数受信します。このようなメッセージ，特にコントロールに送られるメッセージに対し，Windowsではデフォルト応答を用意しています。たとえば，ユーザがエディットコントロールにタイプすると，Windowsはデフォルト応答

として，新しく入力された文字を表示し，必要であればスクロールします。Windowsのデフォルト応答は，ダイアログオブジェクトやコントロールオブジェクトなどのObjectWindowsのウィンドウオブジェクトに対して行なわれます。メンバ関数 ***DefWndProc*** がこのデフォルト応答を起動します。

Windowsメッセージを無視すると（すなわち，応答用のメンバ関数を定義しないと），ObjectWindowsがインターフェースオブジェクトのメンバ関数 ***DefWndProc*** を自動的に呼び出します。ただし，メッセージ応答メンバ関数を定義してメッセージを受け取る場合でも，明示的に ***DefWndProc*** を呼び出すことがあります。

たとえば，以下のコード例では ***WM_CHAR*** メッセージを受け取ります。ユーザがエディットコントロールにタイプすると，このメッセージがエディットコントロールオブジェクトに送出されます。次に示すのは，***TEdit*** からの派生クラスに対する ***WM_CHAR*** メッセージへの応答メンバ関数です。

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
}
```

このメンバ関数は，ユーザのタイプ入力に対してビープ音を鳴らします。ただし，このように応答を定義すると，デフォルトのウィンドウ手続きの標準の応答が抑止されてしまいます。したがって，ビープ音は聞こえますが，エディットコントロールのテキストは変更されません。

ビープ音を鳴らして，しかも標準応答も行なわせるには，次のように明示的にメンバ関数 ***DefWndProc*** を呼び出します。

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

このメンバ関数では，ビープ音が鳴るだけでなく，エディットコントロールに文字も追加されます。

一般に，***DefWndProc*** を呼び出すのは次のような場合です。

- メッセージの受信を確認するが，デフォルトのメッセージ処理をオーバーライドしない場合。
- メッセージを受け取り，標準応答以外の処理を行なわせるが，標準応答も行なわせたい場合。
- メッセージを受け取ると使えなくなる機能がある場合。

第10章

ウィンドウオブジェクト

この章では，アプリケーションのウィンドウを作成・表示し，中味を描画する方法について説明します。ObjectWindows のクラス ***TWindow*** でプログラムのメインウィンドウとその他のウィンドウの基本的な動作の大部分を定義します。***TWindow*** ではウィンドウのオープン，クローズ，描画，スクロールなどの動作，および子ウィンドウの動作，コマンドメッセージの処理を定義します。

TWindow には ***TMDIFrame***，***TMDIClient***，***TControl***，***TEditWindow*** という 4 つの派生クラスがあります。***TMDIFrame*** と ***TMDIClient*** は，Windows のマルチドキュメントインターフェース（MDI）に準拠した ObjectWindows アプリケーションで使います。MDI とこの 2 つのクラスについては第 14 章を参照してください。***TControl*** は，ボタンやリストボックスなどのコントロールを定義します。説明は第 12 章にあります。この章で説明する ***TFileWindow*** は，***TEditWindow*** からの派生クラスです。多くの場合，新しく作成するウィンドウクラスは ***TWindow*** から派生させることになります。

この章では ***TWindow***，***TEditWindow***，***TFileWindow*** を説明します。新しい Windows クラスを登録する方法も例を使って説明します。

TWindow クラス

ObjectWindows アプリケーションには，起動時に画面に表示されるメインウィンドウが最低限必要です。ObjectWindows プログラムのメインウィンドウは通常，***TWindow*** から派生させたクラスのインスタンスです。また，たいていのアプリケーションには，メインウィンドウの子ウィンドウとなる別のウィンドウがあります。このようなウィンドウも，通常 ***TWindow*** の派生クラスのインスタンスです。たとえば，ペイントプログラムでは *TPaintWindow* クラスをメインウィンドウとして定義し，描画の拡大表示を行なう *TZoomWindow* クラスを別のウィンドウとして定義することがあります。この場合，*TPaintWindow* と *TZoomWindow* はどちらも ***TWindow*** の下位クラスです。うまく設計されたアプリケーションでは，*TZoomWindow* は *TPaintWindow* を専用化したものになり，*TPaintWindow* の派生クラスとして実現されます。したがって，間接的に ***TWindow*** の派生クラスになります。

ウィンドウオブジェクトの初期化と作成

ダイアログオブジェクトやコントロールオブジェクトと同様，ウィンドウオブジェクトはビジュアルインターフェース要素に対応するインターフェースオブジェクトです。すなわち，ウィンドウオブジェクトはウィンドウ要素を表わし，データメンバ ***HWindow*** に格納されるハンドルによって識別されます。ウィンドウには 2 つの部分があるため，その作成手順も 2 段階を踏みます。すなわち，まずオブジェクトを構築し，次にビジュアル要素を作成します。

ウィンドウオブジェクトの初期化

代表的な Windows アプリケーションには，実に様々なスタイルのウィンドウがあります。たとえば，オーバーラップしたりポップアップしたりするもの，枠のあるもの，スクロールできるもの，タイトルを持つものなどがあります。このようなウィンドウのスタイルを決める属性は通常，他の作成属性と同様に，ウィンドウオブジェクトの構築時に設定され，ウィンドウオブジェクトが表わすビジ

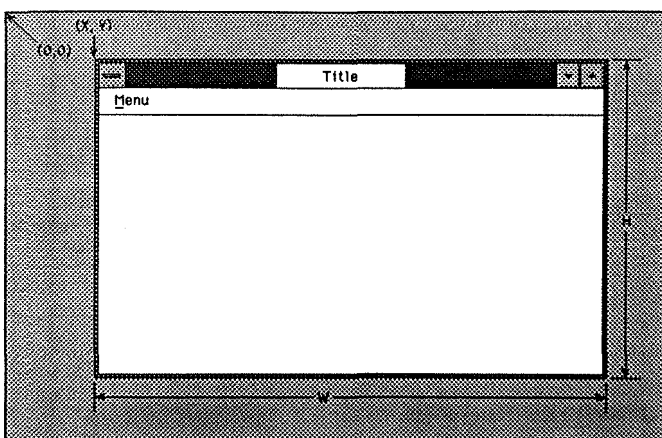
ュアル要素の作成時に使用されます。

スタイル，タイトル，メニューといったウィンドウオブジェクトの作成属性は，***TWindowAttr*** クラスに定義されているデータメンバ ***Attr*** に格納されます。***TWindowAttr*** には次のデータメンバがあります。

表 10.1
TWindowAttr のデータメンバ

データメンバ	内容
Style	スタイル定数を組み合わせて保持する ***DWORD***。
ExStyle	拡張スタイルを保持する ***DWORD***。
Menu	メニューリソースを識別する ***LPSTR***。
X	ウィンドウの初期の横位置を表わす int。ウィンドウの左上隅に対応する水平方向の画面座標。
Y	ウィンドウの初期の縦位置を表わす int。ウィンドウの左上隅に対応する垂直方向の画面座標。
W	画面座標でウィンドウの初期の幅を表わした int。
H	画面座標でウィンドウの初期の高さを表わした int。
Param	ウィンドウの作成時にウィンドウに渡される ***LPSTR***。
Id	子ウィンドウ ID を指定する int。コントロールとその親ウィンドウ間またはコントロールとダイアログ間の通信に使います。同じ親を持つ各子ウィンドウは他と重複しない固有の *Id* が必要です。コントロールがリソースで定義されている場合，リソース ID と同じ *Id* にします。1 つのウィンドウに ***Menu*** と ***Id*** の両方を同時に設定してはいけません。

図 10.1
ウィンドウの属性



コンストラクタ呼び出しの第1引数は，ウィンドウの親ウィンドウオブジェクトです。メインウィンドウの場合は親がないのでヌルにします。***TWindow*** のコンストラクタはデータメンバ ***Title*** に指定された ***LPSTR*** 引数を設定します。ウィンドウがアプリケーションのメインウィンドウであれば，***Attr*** のデータメンバ ***Style*** にもデフォルトで ***WS_OVERLAPPEDWINDOW*** を設定します。メインウィンドウ以外は，***WS_VISIBLE*** を設定します。***X*** と ***W*** には ***CW_USEDEFAULT*** を設定し，***Y*** と ***H*** には0を設定します。これで，適当なサイズのオーバーラップウィンドウになります。これが新しいメインウィンドウのサイズを決める標準的な方法です。

TWindow の派生クラスとしてウィンドウクラスを作成する場合は，少なくとも基本クラスのコンストラクタを呼び出すコンストラクタを次のように定義しなければなりません。

```
class MyWindow : public TWindow
{
⋮
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) : TWindow(AParent,
            ATitle) {}
⋮
};
```

これで，コンストラクタのデータメンバ ***Attr*** を直接変更して，新しいウィンドウオブジェクトの属性を設定し直すことができます。たとえば，次のようにウィンドウを最大化して表示させることもできます。

```
class TMyWindow : public TWindow
{
⋮
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
⋮
  PTChildWindow AChildWindow;
  PTListBox ListBox;
};
⋮
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                  TWindow(AParent, ATitle)
{
  Attr.Style |= WS_MAXIMIZE;
  AChildWindow = new TChildWindow(this, "Child Title");
  ListBox = new TListBox(this, ID_LISTBOX, 201, 20, 20, 180, 80);
⋮
}
```

上記の例では，サンプルウィンドウのコンストラクタがポップアップウィンドウやリストボックスなどの子ウィンドウオブジェクトを構築します。構築された子ウィンドウはその属性を，自分のコンストラクタで設定し直せます。

```
TChildWindow::TChildWindow(PTWindowsObject AParent, LPSTR ATitle) :
                              TWindow(AParent, ATitle)
{
  Attr.Style |= WS_POPUP | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
  Attr.H = 300;
}
```

Style 属性は，次のウィンドウスタイル定数を単独で指定することも，複数を組み合わせることもできます。

- ***WS_OVERLAPPEDWINDOW***
- ***WS_POPUPWINDOW***
- ***WS_CHILDWINDOW***
- ***WS_CAPTION***
- ***WS_BORDER***
- ***WS_VSCROLL***

これは一般的なスタイル定数です。詳細については，**ObjectWindows for C++リファレンスガイド**の第7章『**Windows 定数**』を参照してください。

子ウィンドウに対して派生クラスを定義しない方法もあります。すなわち，ウィンドウオブジェクトを生成してから，その属性を設定し直すというやり方です。この作業は，すべて親ウィンドウのコンストラクタの内部からできます。

Attr.Menu メンバを直接設定することはできません。かわりにメンバ関数 AssignMenu を使ってください。

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                        TWindow(AParent, ATitle)
{
  AssignMenu(100);
  AChildWindow = new TWindow(this, "Child Title");
  AChildWindow->Attr.Style = WS_POPUP | WS_CAPTION;
  AChildWindow->Attr.X = 100;
  AChildWindow->Attr.Y = 100;
  AChildWindow->Attr.W = 300;
  AChildWindow->Attr.H = 300;
  ⋮
}
```

ウィンドウ要素の作成

ウィンドウ要素の作成とは，ウィンドウオブジェクトに対応するビジュアル要素を構築する処理のことです。ウィンドウオブジェクトを引数としてアプリケーションオブジェクトのメンバ関数 ***MakeWindow*** を呼び出します。

GetApplication()はアプリケーションオブジェクトへのポインタを返します。

```
if (GetApplication() -> MakeWindow(AWindow))
  // 作成成功
else
  // 作成失敗
```

MakeWindow はさらに，2つの重要なメンバ関数を呼び出します。最初のメンバ関数は ***ValidWindow*** で，ウィンドウオブジェクトが正常に構築されたかどうかを調べるものです。何らかの理由でコンストラクタが正常に動作しなかった場合，***MakeWindow*** はそのウィンドウオブジェクトを削除してヌルを返します。コンストラクタが正常に動作すると，***MakeWindow*** はウィンドウオブジェクトのメンバ関数 ***Create*** を呼び出します。この ***Create*** は，ビジュアル要素の作成を実際に Windows に指示するメンバ関数です。この場合も，***Create*** が処理に失敗すると，***MakeWindow*** はウィンドウオブジェクトを削除してヌルを返します。成功すると，ウィンドウオブジェクトを指すポインタを返します。

実際にウィンドウ要素を作成するメンバ関数は ***Create*** ですが，通常は明示的に ***Create*** を呼び出しません。アプリケーションのメインウィンドウは，起動時に ***TApplication*** :: ***InitInstance*** によって自動的に作成されます。他のアプリケーションウィンドウは，直接的であれ間接的であれ，すべてメインウィンドウの子ウィンドウになります。そして，子ウィンドウは通常，親ウィンドウオブジェクトのメンバ関数 ***SetupWindow*** で作成されます。ただし，***MakeWindow*** を呼び出してウィンドウを動的に作成することもできます。

初期化と作成のまとめ

新しいウィンドウオブジェクトクラスを設計するとき，そのコンストラクタをどう定義するかによって，ウィンドウの属性を設定させたり，子ウィンドウオブジェクトを構築させたりすることができます（ウィンドウ属性は親ウィンドウのコンストラクタでも設定できます)。

GetApplication()->MakeWindow はウィンドウのメンバ関数 ***Create*** を呼び出してウィンドウオブジェクトのビジュアル要素を作成します。そのデータメンバ *Attr.Style* にスタイル ***WS_VISIBLE*** が設定してあれば，ウィンドウは作成と同時に表示されます。その後でウィンドウのメンバ関数 ***SetupWindow*** が呼び出され，デフォルトで子ウィンドウを作成します。

親ウィンドウのコンストラクタは，通常その子ウィンドウを構築します。子ウィンドウオブジェクトの属性は，通常子ウィンドウ自身のコンストラクタか親ウィンドウのコンストラクタによって設定されます。なお，アプリケーションのメインウィンドウには親がないため，アプリケーションの起動時にアプリケーションオブジェクトが構築し作成します。

ウィンドウクラスの登録

メンバ関数 AssignMenu を呼び出してウィンドウのメニューを更新できます。

これまでの説明で，ウィンドウオブジェクトを初期化するときに，ウィンドウのスタイルや位置といった属性を設定できることがわかりました。具体的には，オブジェクトのデータメンバ ***Attr*** を設定するわけです。このような属性は，対応するウィンドウ要素を作成するときに使われるため**作成属性**といいます。

これ以外にも，バックグラウンド色，表示用アイコン，マウスカーソルなどの属性があります。Windows でいくつかのアプリケーションを同時に実行してみればこのような属性を実際に見ることができます。マウスカーソルをあるアプリケーションから別のアプリケーションに移動してみてください。移動先のアプリケーションがスプレッドシートであれば，通常の矢印のマウスカーソルは太い十字に変化し，移動先がグラフィックスプログラムであれば，細い十字に変化します。

このようなアイコンなどのウィンドウ属性は，ウィンドウの作成時に**ウィンドウクラス**を特定することによって設定されます。このように Windows クラスに対して定義された属性は，作成属性とは異なり，ウィンドウごとに変更することはできません。同一のウィンドウクラスに属するすべてのウィンドウに対して付与されるものだからです。

☞ ウィンドウの Windows クラスは ObjectWindows のウィンドウオブジェクトのクラスとは異なります。ただし，ObjectWindows のウィンドウクラスの各インスタンスは，同じ Windows クラスを共有します。

Windows クラスを**登録**していないと，その Windows クラスのウィンドウを作成することができません。実際の登録は，ウィンドウの作成時に ObjectWindows が行ないます。さらに，ObjectWindows ではデフォルトの**登録属性**のついたデフォルトの Windows クラスが提供されています。このデフォルトを変更をしない限り，登録の手間はいっさいかかりません。

ウィンドウのこのデフォルト属性を変更する場合は，新しい Windows クラスを定義してそのウィンドウを対応させなければなりません。この場合，***GetWindowClass*** と ***GetClassName*** の 2 つの仮想メンバ関数を定義し直します。***GetWindowClass*** では必要な登録属性を特定します。また，***GetClassName*** を定義し直して，その Windows クラスの名前を決めます。***GetClassName*** はウィンドウクラスの名前（***LPSTR***）を返すだけの関数です。***TWindows*** で定義されているメンバ関数 ***GetClassName*** は，デフォルトのウィンドウクラス名である「OWLWindow」を返します。

標準の矢印のかわりに，大文字の I の形をしたカーソルを使う *IBeamWindow* というウィンドウオブジェクトクラスを定義する場合を考えてみましょう。この場合，継承したメンバ関数 ***GetClassName*** を次のように定義し直します。

```
LPSTR IBeamWindow::GetClassName()
{
  return "IBeamWindow";
}
```

返されるクラス名は Windows クラスの識別子として Windows での登録時に使われます。

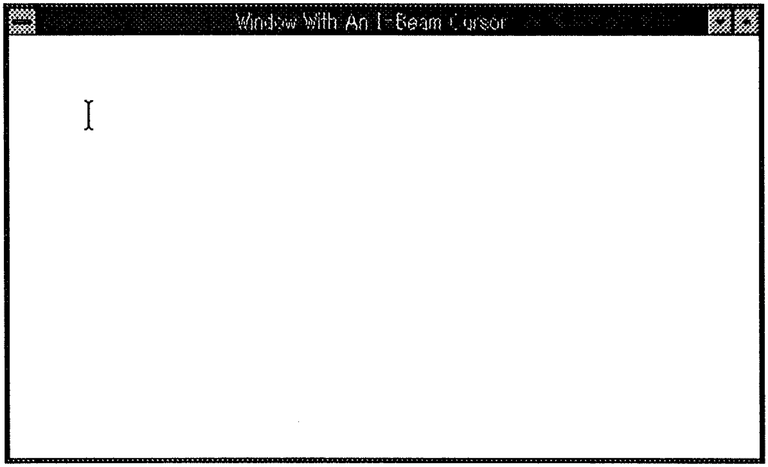
返されるクラス名は，**実際のクラス名**と同じでも別のクラス名でもかまいません。

メンバ関数 ***GetWindowClass*** は ***WNDCLASS*** 構造体を参照引数として使い，各フィールドに新しい登録属性を入れます。最初に基本クラスのメンバ関数 ***GetWindowClass*** を呼び出し，次に，変更するフィールドを設定します。以下の例では，***TWindow*** : : ***GetWindowClass*** を呼び出し，次に，カーソルリソースのハンドルを格納する ***hCursor*** フィールドを設定し直しています。

```
void IBeamWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(NULL, IDC_IBEAM);
}
```

IDC_BEAM は，Windows のストックカーソルを表わす定数です。*IBeamWindow* オブジェクトを使ったアプリケーションを**図 10.2** に示します。

図 10.2
IBeamWindow クラスを使ったプログラム



登録属性

ObjectWindows には，***TWindow*** の各インスタンスに使うデフォルトの Windows クラスが 1 つ定義されています。この「OWLWindow」の登録属性には，空白のアイコン，矢印カーソル，標準のウィンドウ色があります。***TWindow*** から派生させたクラスでこれらの特性を変える場合は，メンバ関数 ***GetClassName*** を定義し直す必要があります。***GetWindowClass*** には ***WNDCLASS*** 構造体への参照が渡されるので，この構造体に登録属性を設定しなければなりません。

WNDCLASS 構造体の特性フィールドの一部と ***TWindow::GetWindowClass*** で設定されるそれぞれのデフォルト値を次の表に示します。

表 10.2
ウィンドウ登録属性

特性	フィールド	デフォルト
クラススタイル	***style***	***CS_HREDRAW*** | ***CS_VREDRAW***
アイコン	***hIcon***	***LoadIcon*** (0, ***IDI_APPLICATION***)
カーソル	***hCursor***	***LoadCursor*** (0, ***IDC_ARROW***)
バックグラウンド色	***hbrBackground***	(***HBRUSH***)(***COLOR_WINDOW*** +1)
デフォルトメニュー	***lpszMenuName***	NULL

クラススタイルメンバ

WNDCLASS 構造体のメンバ ***style*** は，ウィンドウオブジェクトの作成属性として ***Attr.Style*** で指定したウィンドウスタイルとは異なります。***Attr.Style*** はウィンドウが作成されたときの外観を指定するのに使い，***WS***（ウィンドウスタイル）定数で設定します。

WNDCLASS 構造体のメンバ ***style*** は，ウィンドウが作成されてからのウィンドウの動作を指定するもので，***CS_***（クラススタイル）定数で設定します。たとえば，***CS_HREDRAW*** が設定されていると，ウィンドウの横方向の大きさが変わったときにウィンドウ全体が描画し直されます。***CS_NOCLOSE*** が設定されていると，コントロールメニューの Close オプションが使えません。この *style* には，***CS_*** 定数を単独でも，組み合わせたものでも設定できます。***CS_*** 定数の一覧については，**ObjectWindows for C++リファレンスガイド**の第7章『**Windows 定数**』を参照してください。

TWindow::GetWindowClass は，このメンバを次のように設定します。

```
AWndClass.style = CS_HREDRAW | CS_VREDRAW;
```

アイコンメンバ

メンバ ***hIcon*** は，最小化されたウィンドウを表わすアイコンへのハンドルを保持します。通常，プログラムのメインウィンドウを表わすアイコンリソースを定義します。***TWindow::GetWindowClass*** はこのメンバに，空白の矩形として表示されるストックアイコン ***IDI_APPLICATION*** へのハンドルを設定します。

カーソルメンバ

データメンバ ***hCursor*** は，ウィンドウ上に位置づけられたマウスポインタを表わすカーソルへのハンドルを保持します。***TWindow::GetWindowClass*** はこのメンバに，Windows 標準の矢印 ***IDC_ARROW*** のハンドルを設定します。その他のストックカーソルには次のものがあります。

- *IDC_IBEAM*
- *IDC_WAIT*
- *IDC_CROSS*

アイコンなどのカーソルはユーザが定義できます。

バックグラウンド色メンバ

ウィンドウのバックグラウンド色を指定します。***TWindow*** :: ***GetWindowClass*** はこのメンバに，デフォルトのシステムウィンドウカラーを設定します。このメンバの設定はコントロールパネルから行なえます。このメンバに特定の色の物理ブラシへのハンドルを設定することもできます。あるいは，***hbrBackground*** に ***COLOR_ACTIVECAPTION*** +1 を設定すれば，***COLOR_ACTIVECAPTION*** のようなシステムカラーを設定することもできます。

デフォルトメニューメンバ

このメンバには，当該 Windows クラスのすべてのウィンドウが共有するメニューのリソース名のポインタが設定されます。たとえば，つねに標準のエディットメニューを持つ ***EditWindow*** 型を定義する場合には，そのメニューをこのメンバで指定します。こうしておけば，コンストラクタでメニューを作成属性として指定する手間が省けます。メニューリソースのリソース ID が 101 である場合，このメンバの設定は次のようになります。

```
AWndClass.lpszMenuName = MAKEINTRESOURCE(101);
```

ウィンドウのスクロール

ウィンドウに表示されるのは一部の内容です。ObjectWindows のウィンドウは簡単なプログラミングで，ウィンドウをスクロールさせて内容全体を見ることができます。***TWindow*** は ***TScroller*** オブジェクトを使ってこのスクロールを実現します。

ほとんどの場合，ユーザはウィンドウのクライアント領域の端にあるスクロールバーを操作して，表示範囲をスクロールします。このウィンドウスクロールバーはコントロールではなく，ウィンドウに組み込まれたものです。作成スタイルとして ***WS_VSCROLL*** か ***WS_HSCROLL*** が指定されているウィンドウに作成されます。これらのウィンドウスクロールバーを実際に動かすのが ***TScroller*** で，自動的なスクロールを可能にします。

また，ObjectWindows のウィンドウは，ユーザがクライアント領域の内側から外側にマウスをドラッグしたときもスクロールさせることができます。これを自動スクロールといいます。スクロールバーのないウィンドウでも自動スクロールは可能です。

スクローラの属性

TScroller は，ウィンドウのスクロール量を決める数値を保持します。数値は ***TScroller*** のデータメンバ ***XUnit***，***YUnit***，***XLine***，***YLine***，***XPage***，***YPage***，***XRange***，***YRange*** に格納されます。***X*** で始まるデータメンバは横方向の値を示し，***Y*** で始まるデータメンバは縦方向の値を示します。

スクロールの最小単位をスクロール単位といいます。これは，縦または横にスクロールするときの最小の物理単位です。一般にはウィンドウのピクセル数で表わしますが，使用中のマッピングモードによって異なります。スクロール単位は，表示する情報の種類によっても異なります。たとえば，表示するテキストの平均文字幅が 8 ピクセルで高さが 15 ピクセルの場合，***XUnit*** と ***YUnit*** の値はそれぞれ 8 と 15 になります。

行，ページ，範囲のスクロール属性は，スクロール単位をもとにして表わされます。***Line*** と ***Page*** の値は，ユーザからのスクロール要求に応じてスクロールする単位数を表わします。ユーザがスクロールバーの端にあるいずれかの矢印をクリックすると，ウィンドウは行単位でスクロールします。この行単位数が行データメンバに格納されます。また，スクロールボックスではなく，スクロールバー自体をクリックすると，ページ単位でスクロールします。範囲属性は総スクロール単位数を表わし，通常，編集中の書類などウィンドウが見せている世界の大きさに対応します。

それでは，例としてテキスト編集ウィンドウを見てみましょう。表示しようとするファイルは 400 行のテキストで，1 行の文字数が 80 文字，1 ページの行数が 50 行であるとします。このとき，各データメンバの値は次のようになります。

表 10.3
代表的な編集ウィンドウのデータメンバの設定値

データメンバ	値	意味
XUnit	8	文字の幅
YUnit	15	文字の高さ
XLine, YLine	1	1 単位(Unit)/行
XPage	40	40 文字/ページ
YPage	50	50 行/ページ
XRange	80	最大横範囲
YRange	400	最大縦範囲

これらの値を設定した ***TScroller*** オブジェクトは，1行ずつまたは1ページずつスクロールします。ファイル全体を見るには，スクロールバーを使うか自動スクロールを行ないます。

デフォルトの行数値は1です。特別な値を使う場合以外はこの値を設定し直す必要はありません。デフォルトのページ単位は現在のウィンドウのサイズによって決まります。つまり，ページ単位のスクロールとは，現在のクライアント領域の高さと幅を単位にしていずれかの方向にスクロールするということです。ページの数値も特別な値が必要でなければ変更する必要はありません。

ウィンドウにスクローラを追加する

ウィンドウにスクローラを追加するには，ウィンドウのコンストラクタで ***TScroller*** オブジェクトを生成して，このオブジェクトへのポインタをウィンドウのデータメンバ ***Scroller*** に格納します。単位や範囲の値も同時に設定できます。自動スクロールの場合のように，スクローラを使うときにスクロールバーは必ずしも必要ではありませんが，スクロールできるウィンドウではたいてい使われています。スクロールバーをウィンドウに追加するには，コンストラクタのデータメンバ ***Attr.Style*** に ***WS_VSCROLL*** と ***WS_HSCROLL*** のどちらかまたは両方を追加します。テキスト編集ウィンドウの例に使うコンストラクタは次のようになります。

このコードは SCROLAPP.CPP の一部です。

```
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}
```

TScrollerのコンストラクタは，スクロール可能ウィンドウの値とデータメンバ ***XUnit***, ***YUnit***, ***XRange***, ***YRange*** の初期値を引数としてとります。行とページの属性はデフォルト値に設定されます。

このウィンドウを表示してみると，クライアント領域の内容は，スクロールバーや自動スクロールを使って横方向や縦方向にスクロールすることができます。ウィンドウのメンバ関数 ***Paint*** はグラフィックス情報を描画するだけでよく，スクロールが行なわれたかどうかを意識する必要はありません。一部しか表示されない大きな絵の全体を描画するのは非効率的なので，メンバ関数 ***Paint*** を最適化すれば，絵の表示される部分だけを描画させることができます。***Paint*** の最適化についてはこの節の最後で説明します。

スクロールの例

ここでは，スクロール可能なグラフィックスを描画するアプリケーションの構築方法を例を使って説明します。このアプリケーションは，大きさが徐々に大きくなっていく一連の四角形を描画するものです。そのため，通常の画面に表示したウィンドウのクライアント領域では絵の全体を一度に見ることはできません。そこで，絵の隠れている部分を見るために，スクロールバーや自動スクロールを使うことになります。自動スクロールを行なうには，クライアント領域でマウスの左ボタンを押して，そのままカーソルを領域の外に移動させます。

SCROLAPP.CPP

```
#include <owl.h>

// TApplication の下位である TScrollApp を宣言
class TScrollApp : public TApplication {
public:
  TScrollApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
             LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
             hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

// TWindow の下位である TScrollWindow を宣言
class TScrollWindow : public TWindow
{
public:
  TScrollWindow(LPSTR ATitle);
  virtual void Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo);
};
```

```
/* TScrollWindow 用のコンストラクタです. スクロールスタイルを設定して,
  Scroller オブジェクトを構築します. */
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}

/* ボックスを描画しなおして, 送られてくる "paint" メッセージに応答します.
  現在のスクロール位置を基準にしてビューポートの原点を設定する Scroller
  の BeginView メンバ関数がすでにコールされていることに注意してください. */
void TScrollWindow::Paint(HDC PaintDC,PAINTSTRUCT&)
{
  int X1, Y1, I;

  for (I = 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
    Y1 = 30 + I*5;
    Rectangle(PaintDC, X1, Y1, X1 + X1, X1 + Y1 * 2);
  }
}

// TScrollWindow 型の TScrollApp の MainWindow を構築
void TScrollApp::InitMainWindow()
{
  MainWindow = new TScrollWindow("Boxes");
}

// ScrollApp を実行
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TScrollApp ScrollApp("ScrollApp", hInstance, hPrevInstance,
                       lpCmdLine, nCmdShow);
  ScrollApp.Run();
  return ScrollApp.Status;
}
```

自動スクロールとトラッキング

TScroller の自動スクロール機能はデフォルトで利用できるようになっていますが，***TScroller*** のデータメンバ ***AutoMode*** を偽に設定すれば使えなくなります。この設定は，***TScroller*** オブジェクトを構築した後，オーナーウィンドウのコンストラクタで行ないます。次のように指定します。

```
Scroller = new TScroller(this, 8, 15, 80, 60);
Scroller->AutoMode = FALSE;
```

このように設定すると，スクロールバーでのみスクロールできるようになります。

自動スクロール機能の長所の1つは，クライアント領域からマウスを遠ざけるほど，ウィンドウのスクロールが速くなるという点です。マウスをどれだけ遠ざけるかによって，ウィンドウのスクロール単位は行値まで小さくなったり，ページ値まで大きくなったりします。

自動スクロールに加えて，上記のプログラムでは，スクロールボックスの移動によるスクロール要求も追跡しています（トラッキング)。つまり，スクロールボックスの移動につれて絵が移動するわけです。この機能を使うと結果が即座に現われるため，マウスボタンを押したまま希望する絵の部分を見ることができます。

トラッキングが不都合な場合もあります。たとえば，大きなテキストファイルを表示しているときは，スクロールボックスを動かすたびにディスクからの読み出しが行なわれ，スクロールボックス位置のテキストが表示されます。その結果，トラッキングが遅くなってしまいます。このような場合には，次のようにしてトラッキングをオフにします。

```
Scroller->TrackMode = FALSE;
```

このように設定しておくと，スクロールボックスを動かしてもマウスボタンを離さない限りスクロールされません。マウスボタンを離せば，クライアント領域は1回だけスクロールして希望する絵の部分が表示されます。

スクロール単位と範囲の変更

これまでの例では，単位と範囲の値は，***TScroller*** の構築時にわかっているものと仮定してきました。しかし，多くの場合，構築時にはこの情報はわからず，また変更することもあります。たとえば，表示する情報の大きさが変わった場合などは，範囲値や単位値の設定や変更が必要になります。構築時に値がわからないときは，0を ***TScroller*** のコンストラクタに渡してください。

メンバ関数 ***SetRange*** には，スクロール全体の範囲を決める水平単位数と垂直単位数という2つのlong引数を渡します。絵のサイズを変えるときは，必ず ***SetRange*** を使ってください。たとえば，横100ユニット，縦300ユニットの絵を表示する場合，次のコマンドによってスクロール範囲を設定できます。

```
Scroller->SetRange(100, 300);
```

TScroller オブジェクトを初期化するときに単位値がわからなかった場合は，スクロールが実行されるまでに値を設定してください。たとえば，ウィンドウのメンバ関数 ***SetupWindow*** で次のように設定できます。

```
void ScrollWindow::SetupWindow()
{
  Twindow::SetupWindow();
  Scroller->XUnit = 10;
  Scroller->YUnit = 20;
}
```

スクロール位置の変更

TScroller のメンバ関数 ***ScrollTo*** と ***ScrollBy*** を呼び出して強制的にスクロールさせることができます。両メンバ関数にそれぞれ，水平と垂直のスクロール単位を表わす2つの long 引数を渡します。たとえば，スクロール位置を絵の左上隅に設定し直したい場合には，***ScrollTo*** を次のように使います。

```
Scroller->ScrollTo(0, 0);
```

また，垂直方向に400単位の長さを持つ絵の場合，次のようにすればスクロール位置を絵の中央にセットすることができます。

```
Scroller->ScrollTo(0, 200);
```

メンバ関数 ***ScrollBy*** は，単位数に応じて上下左右にスクロール位置を移動します。負の値を指定すると，スクロール位置は基点すなわち左上隅の方向に向かって移動します。正の値を指定すれば右と下の方向に移動します。たとえば，右に10単位，下に20単位だけスクロールしたい場合は次のコマンドを使います。

```
Scroller->ScrollBy(10, 20);
```

ページサイズの設定

デフォルトのページサイズ（***XPage*** と ***YPage***）は，ウィンドウのクライアント領域のサイズによって決まります。スクローラのオーナーウィンドウのサイズが変わると，スクローラに通知されます。まず，オーナーウィンドウのメンバ関数 ***WMSize*** がスクローラのメンバ関数 ***SetPageSize*** を呼び出します。呼び出されたメンバ関数は，クライアント領域の現在のサイズおよび ***XUnit*** と ***YUnit*** の値に基づいて，データメンバの ***XPage*** と ***YPage*** を設定します。

この方法を定義し直して，ページサイズを直接設定することもできます。この場合は，ウィンドウオブジェクトが継承したメンバ関数 ***WMSize*** を次のように定義し直します。

```
void TTestWindow::WMSize(RTMessage Msg)
{
  TWindow::WMSize(Msg);
}
```

このように設定すると，ウィンドウのコンストラクタから，または ***TScroller*** の派生クラスのコンストラクタから，直接 ***XPage*** と ***YPage*** を設定することができます。

```
TScrollWindow::TScrollWindow(PTWindowsObject AParent, LPSTR ATitle) :
                              TWindow(AParent, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 400);
  Scroller -> XPage = 40;
  Scroller -> YPage = 100;
}
```

メンバ関数 Paint の最適化

上のアプリケーションは 50 の矩形を描画しますが，すべての矩形がウィンドウのクライアント領域に実際に表示されるかどうかは判断していません。そのため，見えないグラフィックスまでも描画しようとして処理に無駄が生じてしまいます。このような場合，ウィンドウのメンバ関数 ***Paint*** で ***TScroller::IsVisibleRect*** 関数を使うと，描画処理を最適化することができます。

以下に示すメンバ関数 ***ScrollWindow* :: *Paint*** は，***IsVisibleRect*** を使ってWindows 関数 ***Rectangle*** を呼び出すかどうかを決定します。***Rectangle*** の引数は物理単位になっています。これに対して，***IsVisibleRect*** の引数はスクロール単位となっています。このため，***IsVisibleRect*** を呼び出す前に，矩形の基点を表わす *X1* と *Y1*，矩形の幅と高さを表わす（*X2* - *X1*）と（*Y2* - *Y1*）を，それぞれの単位値で除算しておく必要があります。

```
void TscrollWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  int X1, Y1, X2, Y2, I;

  for (I = 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
    Y1 = 30 + I*5;
    X2 = X1 + X1;
    Y2 = X1 + Y1 * 2;
    if (Scroller->IsVisibleRect(X1/Scroller->XUnit,
                                Y1/Scroller->YUnit,
                                (X2-X1)/Scroller->XUnit,
                                (Y2-Y1)/Scroller->YUnit))
      Rectangle(PaintDC, X1, Y1, X2, Y2);
  }
}
```

Edit ウィンドウと File ウィンドウ

30ページの「ObjectWindows アプリケーションの構築」を参照してください。

ObjectWindows には ***TWindow*** の下位型が 2 つあり，テキスト編集専用のウィンドウとして使えます。***TEditWindow*** クラスのインスタンスは，ファイルの読み書きができないテキスト編集ウィンドウです。一方，***TEditWindow*** の派生クラスである ***TFileWindow*** は，ファイル I/O に必要な動作を追加定義します。両クラスのインスタンスは作成したアプリケーションでそのまま使うこともできますが，両クラスから専用のテキストエディタを派生させて作成することもできます。Edit ウィンドウや File ウィンドウを使用するプログラムは，それぞれ EDITWND.H と FILEWND.H をインクルードしておく必要があります。さらに ObjectWindows で提供されているリソースも必要に応じて取り込んでおく必要があります。

Edit ウィンドウ

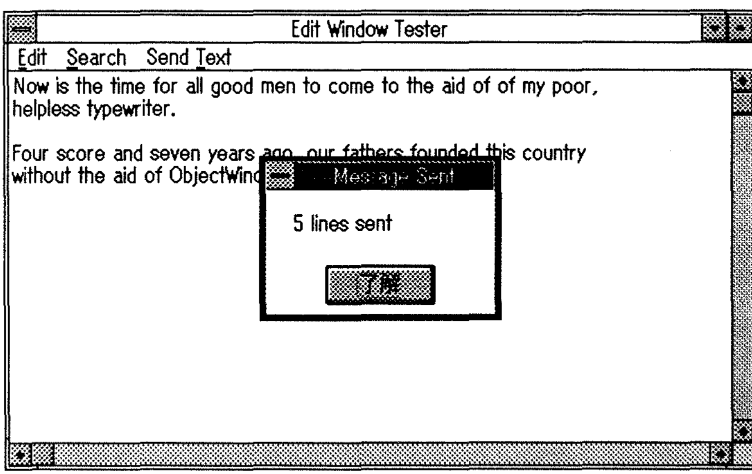
TEditWindow のテキストは，実際は子の ***TEdit*** コントロールで保持され処理されます。処理の大部分を担うのは，ObjectWindows から広範なテキスト編集機能を授かった ***TEdit*** コントロールです。

TEditWindow のコンストラクタは構築した ***TEdit*** のインスタンスを参照するためにデータメンバ ***Editor*** を設定します。メンバ関数 ***WMSize*** は ***TEditWindow*** のサイズが変更されると必ず呼び出され，***TEditWindow*** のクライアント領域が埋まるように ***Editor*** のサイズを変更します。メンバ関数 ***WMSetFocus*** は ***TEditWindow*** にフォーカスが割り当てられると必ず ***TEditor*** に入力フォーカスを渡します。

TEditWindow は，***TEdit*** に組み込まれた基本的な動作 ***Search*** に付加機能を追加します。***TEditWindow*** のユーザに提供される多くの検索/置換機能も担当します。***TEdit*** コントロールは内部のテキストを検索しますが，テキストを置換するのは ***TEditWindow*** です。さらに，***TEditWindow*** は ***TSearchStruct*** を通じて「next」機能も提供します。また，検索/置換のユーザインターフェースを全面的に担当します。

次の EWNDTEST.CPP プログラムは，Edit ウィンドウを使って，機能なしの単純な電子メールアプリケーション用のテキストをユーザに編集させるものです。このアプリケーションの外観を図 **10.3** に示します。

図 10.3
Edit ウィンドウの例



EWNDTEST.CPP

```
#include <owl.h>
#include <stdio.h>
#include <editwnd.h>
#include "ewndtest.h"

class TTestApp : public TApplication
{
public:
  TTestApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
           lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
           hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
  virtual void InitInstance();
};

class TTestWindow : public TEditWindow
{
public:
  TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
  virtual void CMSendText(RTMessage Msg)
    = [CM_FIRST + CM_SENDTEXT];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle) :
                         TEditWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
}

void TTestWindow::CMSendText(RTMessage)
{
  int Lines;
  char Text[20];

  Lines = Editor->GetNumLines();
  sprintf(Text, "%d lines sent", Lines);
  MessageBox(HWindow, Text, "Message Sent", MB_OK);
}

void TTestApp::InitMainWindow()
{
  MainWindow = new TTestWindow(NULL, Name);
}

void TTestApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(hInstance, "EDITCOMMANDS");
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
```

```
{
  TTestApp TestApp("Edit Window Tester", hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

ESNDTEST.RC

```
#include "windows.h"
#include "ewndtest.h"
#include "owlrc.h"

rcinclude stdwnds.dlg
rcinclude editacc.rc

COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&Edit"
  BEGIN
    MenuItem  "&Undo¥aAlt+BkSp", CM_EDITUNDO
    MenuItem  SEPARATOR
    MenuItem  "&Cut¥aShift+Del", CM_EDITCUT
    MenuItem  "C&opy¥aIns", CM_EDITCOPY
    MenuItem  "&Paste¥aShift+Ins", CM_EDITPASTE
    MenuItem  "&Delete¥aDel", CM_EDITDELETE
    MenuItem  "C&lear All¥aCtrl+Del", CM_EDITCLEAR
  END
  POPUP "&Search"
  BEGIN
    MenuItem  "&Find...", CM_EDITFIND
    MenuItem  "&Replace...", CM_EDITREPLACE
    MenuItem  "&Next¥aF3", CM_EDITFINDNEXT
  END
  MenuItem  "Send &Text", CM_SENDTEXT
END
```

File ウィンドウ

TFileWindow のコンストラクタはファイル名を引数にとります。***TFileWindow*** :: ***SetupWindow*** はファイルウィンドウのセットアップとして，***TEditWindow*** から継承した ***TEdit*** コントロールである ***Editor*** にこのファイルの内容を読み込みます。さらに，***TFileWindow*** は ***CanClose*** を定義し直して，ファイルウィンドウをクローズする前に編集内容が残っていれば，ユーザに現在の編集内容を保管するように促します。

TFileWindow には，ユーザの要求に応えてファイルのオープンとセーブを行なう4つのメンバ関数 ***CMFileNew***, ***CMFileOpen***, ***CMFileSave***, ***CMFileSaveAs*** があります。ユーザは，**表 10.4** に示すメニュー項目の識別子を選択して要求を行ないます。

表 10.4
File ウィンドウのメンバ関数とメニュー ID

メンバ関数	呼び出すメニュー ID
CMFileNew	*CM_FILENEW*
CMFileOpen	*CM_FILEOPEN*
CMFileSave	*CM_FILESAVE*
CMFileSaveAs	*CM_FILESAVEAS*

上の要求はいずれもユーザからファイル名を取得するためのものです。***TFileDialog*** のインスタンスを使って，ユーザにドライブ，ディレクトリ，ファイル名を指定させます。

File ウィンドウは，そのままでも単純なスタンドアロンのテキストエディタとして使えますが，***TFileWindow*** から独自のクラスを派生させて別の機能を追加する場合もあります。たとえば，***TFileWindow*** の派生クラスで，***Editor*** に表示するテキストのフォントを変更することができます。これは通常，作成したメニューのドロップダウンサブメニューとしてユーザに提供するようなオプションです。***TFileWindow*** が提供するファイル I/O のユーザインターフェースを使うには，**表 10.4** の識別子を持つメニュー項目を作成する必要があります。

第 11 章

ダイアログオブジェクト

ダイアログは，その作成属性が Windows リソースファイルで指定されるインターフェース要素です。ダイアログのリソース定義で付与された作成属性によって，ダイアログとそのコントロールの外観と配置が指定されます。

ダイアログは通常，子ウィンドウとして使われ，特定の入力に関連するタスクを実行します。たとえば，子ダイアログでユーザからの入力を受け取り，プリンタを設定することができます。ObjectWindows は，***TDialog*** からの派生クラスとして 3 つのクラスを提供します。この 3 つのクラスは共通のタスクを実行します。***TInputDialog*** はユーザが入力した単一行のテキストを取得します。***TFileDialog*** はファイル名を取得します。***TSearchDialog*** はエディットウィンドウのテキストを検索し，置き換えます。

ダイアログリソースの使用

ダイアログインターフェース要素は，必ず Windows リソースファイルでリソースを定義してください。リソースを定義すると固有の識別子がつき，ダイアログ要素とそのコントロールの外観と配置が指定されます。ただし，リソース定義では，ダイアログの動作は指定されません。

リソースを使って ObjectWindows アプリケーションを構築する方法については 30 ページで述べています。リソースは，必ずアプリケーションに付加してください。

ObjectWindows アプリケーションでは，***TDialog*** オブジェクトがダイアログの動作を定義します。このオブジェクトのコンストラクタは，リソース定義の識別子を引数にとります。このリソース定義は，インターフェース要素を作成してこれを ***TDialog*** に対応づけるときに使います。

子ダイアログオブジェクトの使用

子ダイアログオブジェクトの管理は，ポップアップウィンドウの管理とほとんど変わりません。親ウィンドウが存続する間，ダイアログとポップアップウィンドウの作成，表示，破壊は何度も繰り返されます。短時間しか表示させないダイアログとポップアップウィンドウは，1つのメンバ関数によって構築，作成，削除することができ，親ウィンドウのデータメンバとして格納する必要がありません。表示時間が長いものは，親ウィンドウのデータメンバとして格納しておくのが一般的です。

子ダイアログオブジェクトの構築と初期化

TDialog のコンストラクタには2つの引数が必要です。1つは親ウィンドウオブジェクトを指すポインタであり，もう1つはダイアログのリソース識別子です。***TDialog*** はコンストラクタをオーバーロードして，以下のように ***LPSTR*** または int リソース識別子のいずれかをとります。

```
ADialog = new TSampleDialog(this, "SampleDialog");

ADialog = new TSampleDialog(this, 101);
```

識別子には定数を用いることをお勧めします。

```
#define ID_SAMPLERESOURCE 101
⋮
ADialog = new TSampleDialog(this, ID_SAMPLERESOURCE);
```

ダイアログオブジェクトは必ずヒープ上に構築してください。スタックには構築しないでください。Windows 要素をクローズすると，ObjectWindows は自動的に対応するダイアログオブジェクトを削除してしまいます。

ダイアログの作成と実行

子ダイアログオブジェクトとインターフェース要素の対応関係は，***TDialog*** が**実行**または**作成**されたときに成立します。

子ダイアログが**実行**されるのは，ユーザ入力により次の処理が決まるときです。入力を取得し，ダイアログが閉じるまでは処理が中断されます。たとえば，テキスト編集ウィンドウをクローズすると，保管されていない編集内容を保管するかどうかを決定するダイアログが実行されます。編集内容を保管するか破棄するかという次の処理はユーザからの入力により決まります。入力を取得するまでは処理が中断し，ウィンドウはオープンされたままです。ダイアログの実行が終了しないうちは親ウィンドウは使用不可であり，ユーザからの入力に応答できません。

一方，子ダイアログが**作成**されるのは，ユーザ入力があれば次の処理に反映させるが，入力がなければそのまま処理を続行する場合です。たとえば，ユーザが検索を要求したとします。これが初めての検索要求であれば，テキスト編集ウィンドウは検索ダイアログを作成して検索文字列を取得します。このダイアログをユーザが明示的にクローズしないかぎり，以降の検索要求に対して同じダイアログを使えます。この場合，通常のテキスト編集の処理はそのまま続行されます。ダイアログを作成しても，親ウィンドウは使用不可にはなりません。

ダイアログの実行は，ダイアログの作成とは次の点でも異なります。ダイアログを実行した場合，ダイアログはユーザが選択したオプションを示す値を返します。たとえば，単純なダイアログでは，質問のプロンプトを表示することが多く，応答用の OK ボタンと Cancel ボタンがついています。ユーザがどちらかのボタンをクリックすると，ダイアログは終了し，実行結果としてクリックされたボタンの識別子を返します。

第 12 章の節「コントロールデータの転送」を参照してください。

ただし，Windows の場合，ダイアログが実行結果として返せるのは整数値だけです。複雑な入力を取得したい場合は，ObjectWindows が提供するデータ転送機構をカスタマイズするか，ダイアログクラスを派生させて独自に定義した転送バッファを用意し，ここに入力します。

モードつきダイアログとモードなしダイアログ

実行されるダイアログは**モードつき**ダイアログ，作成されるダイアログは**モードなし**ダイアログと呼ばれます。メンバ関数 ***Execute*** と ***Create*** を呼び出して，それぞれモードつきダイアログとモードなしダイアログに ***TDialog*** を対応させることができます。ただし，どちらの関数も直接呼び出すことはできません。モードつきダイアログを安全に**実行**するには，アプリケーションオブジェクトのメンバ関数 ***ExecDialog*** を呼び出し，モードなしダイアログを安全に**作成**するにはメンバ関数 ***MakeWindow*** を呼び出してください。なお，***ExecDialog*** と ***MakeWindow*** は ***TModule*** から派生したものです。

```
// モードつきダイアログを実行
if (GetModule()->ExecDialog(new TSampleDialog(this,
                                "SampleDialog")) == IDOK)
  // なんらかの操作

// モードなしダイアログを作成
GetModule()->MakeWindow(new TSampleDialog(this, "SampleDialog"));
```

子ダイアログのクローズ

224 ページの「コントロールデータの転送」を参照してください。

メンバ関数 ***CloseWindow*** を呼び出して，モードつきダイアログまたはモードなしダイアログをクローズします。この呼び出しは，メッセージ応答メンバ関数 ***Ok*** が行ないます。***CanClose*** がダイアログのクローズを行なえると報告すると，***CloseWindow*** がダイアログをクローズします。モードつきダイアログ用の ***CloseWindow*** は，デフォルトでダイアログのデータを転送します。ダイアログを ***CanClose*** からの許可なしで無条件にクローズしたければ，***ShutDownWindow*** を呼び出します。この呼び出しは，メッセージ応答メンバ関数 ***Cancel*** が行ないます。

228 ページの「データの転送」を参照してください。

ShutDownWindow は通常定義し直しませんが，***CloseWindow*** を定義し直すとモードなしダイアログに ***TransferData*** を呼び出させることができます。さらに，***CanClose*** を定義し直してクローズ前のデータを有効にすることもできます。たとえば，エディットコントロール内のテキストが無効であれば，メッセージボックスを表示させます。

モードつきのダイアログでは，***CloseWindow*** または ***ShutDownWindow*** に渡される値は ***Execute*** の戻り値となり，また ***TModule*** :: ***ExecDialog*** からも返されます。値を特に指定しなければ，デフォルトの ***IDCANCEL*** が戻り値になります。モードなしダイアログには戻り値がないので，値を渡す必要はありません。

ダイアログをメインウィンドウとして使う

モードなしダイアログは，アプリケーションのメインウィンドウとして使えます。特に，メインウィンドウに多数のコントロールを詰め込む場合に便利です。ダイアログをメインウィンドウとして使うと，ウィンドウとそのコントロールの作成属性をダイアログリソースエディタで簡単に定義できます。***TDialog*** からの派生クラスをメインウィンドウとして使ってください。***TWindow*** からの派生クラスの場合と同じように扱い，***InitMainWindow*** で構築することができます。EXAMPLES ディレクトリにある CALC.CPP プログラムでは，モードなしの電卓ダイアログをメインウィンドウとして使っています。

インターフェース要素の指定をアプリケーションから独立させたいときにも，モードなしダイアログをアプリケーションのメインウィンドウとして使うことができます。メインウィンドウの外観を変更するとき，アプリケーションを変更したり再コンパイルする必要がありません。

たとえば，電卓プログラムのメインウィンドウをモードなしのダイアログにすると，電卓ボタンはダイアログリソースのボタンコントロールとなります。この結果，メニュー，アイコン，カーソルをメインウィンドウに追加できます。

ダイアログをメインウィンドウとしたアプリケーションでは，アクセラレータに応答する必要が生じた場合に少しやりにくくなります。Windows がモードなしダイアログでのアクセラレータの使用を完全にはサポートしていないためです。しかし，CALC.CPP プログラムを使えば，この制約を回避できます。

モードなしダイアログ用の Windows クラスの定義

登録属性の概念とウィンドウ属性の定義方法については，第 10 章『**ウィンドウオブジェクト**』の節「**ウィンドウクラスの登録**」ですでに見てきました。要約すると，Windows では，アイコンやカーソルなど特定のウィンドウ属性をウィンドウの作成前に登録しておかなければなりません。ObjectWindows では，ウィンドウのデフォルト属性が登録されます。***GetWindowClass*** と ***GetClassName*** を定義し直せば，このデフォルト属性を変更できます。

Windows ではダイアログの登録属性が前もって定義してあります。モードつきダイアログではこの登録属性を変更できませんが，モードなしダイアログでは変更できます。モードなしダイアログの登録属性を変更するには，ウィンドウの場合と同様に ***GetWindowClass*** と ***GetClassName*** を定義し直します。さらに，ダイアログの Windows クラス名を，ダイアログのリソース定義で指定しなければなりません。このクラス名は，ダイアログのメンバ関数 ***GetClassName*** が返す名前です。

ダイアログのリソース定義で Windows クラス名を指定しないと，Windows で定義されているデフォルトの登録属性がダイアログに付与されます。ダイアログのリソース定義で Windows クラスを指定すると，ダイアログをモードつきで実行できなくなります。

コントロールの操作とメッセージ処理

Windows では，親ウィンドウによるコントロールの作成と管理をほとんどサポートしていません。ObjectWindows アプリケーションでは，コントロールの作成と管理はウィンドウオブジェクトとコントロールオブジェクトが担当します。なお，ウィンドウでのコントロールの使い方については，第 12 章『**コントロールオブジェクト**』を参照してください。

Windows はコントロールの作成をダイアログで処理し，その管理も一部サポートします。したがって，ObjectWindows のデフォルトでは，ダイアログのコントロールはコントロールオブジェクトと対応していません。しかし，ダイアログのコントロールを ObjectWindows のコントロールオブジェクトと対応させて，ダイアログとコントロール間の通信を簡単にすることもできます。

ダイアログオブジェクトとそのコントロール要素は，双方向通信を行ないます。片方向で，ダイアログがコントロールを操作し，照会します。たとえば，ダイアログはリストボックスコントロールに入力したり，エディットコントロールからテキストを取得します。逆方向で，コントロールはイベントの発生をダイアログに通知します。たとえば，ボタンコントロールが押されると，コントロールはその旨を親ダイアログに通知します。

ダイアログコントロールの操作

Windowsには，各種コントロールに対して，一連のコントロールメッセージとデフォルト応答が用意されています。たとえば，***LB_GETCURSEL*** メッセージに対するリストボックスのデフォルト応答では，選択された項目のインデックスが返されます。ダイアログはコントロールに対して，コントロール識別子，メッセージ識別子，メッセージ固有の引数を指定したコントロールメッセージを送ります。

ObjectWindowsではダイアログのコントロールにメッセージを送るとき，ダイアログのメンバ関数 ***SendDlgItemMsg*** を呼び出します。この関数には，次の4つの引数が必要です。

- コントロールの識別子
- メッセージの識別子
- メッセージ（関連データ）の詳細を示す ***WORD*** 型引数
- さらに詳細を示す ***LONG*** 型引数

次のプログラム例では，*TestDialog* のメンバ関数 ***SendDlgItemMsg*** を呼び出し，***LB_ADDSTRING*** メッセージを送ってリストボックスコントロールに"Item 1"を加えます。

```
void TTestDialog::FillListBox()
{
  SendDlgItemMsg(ID_LISTBOX, LB_ADDSTRING, NULL, (DWORD) "Item 1");
}
```

ダイアログクラスのメンバであるコントロールオブジェクトをコントロールに対応づけると，ダイアログオブジェクトとコントロール間の通信が簡単になります。なお前に述べたように，デフォルトではダイアログのコントロールとObjectWindowsのコントロールオブジェクトは対応づけられていません。対応づけてあると，コントロールの操作をオブジェクトのデータメンバと関数メンバを使ってOOPスタイルで行なえます。Windows API経由でコントロールのインターフェース要素にメッセージを送る必要がありません。

この対応づけを行なうには，ダイアログのコンストラクタでコントロールオブジェクトを構築します。親オブジェクトとリソース識別子のみを引数としてとるコントロールオブジェクトのコンストラクタを使ってください。ここで，作成属性を渡す必要はありません。Windowsによって要素はすでに作られているからです。これで，ダイアログのコントロールを，ウィンドウのコントロールのように操作して照会することができます。宣言，コンストラクタ，*TTestDialog*のメンバ関数*FillListBox*に発生する変更を次に示します。

```
class TTestDialog : public TDialog {
public:
 ⋮
  PTListBox ListBox;
  TTestDialog(PTWindowsObject AParent, LPSTR AName);
 ⋮
};

TTestDialog::TTestDialog(PTWindowsObject AParent, LPSTR AName) :
                         TDialog(AParent, AName)
{
  ListBox = new TListBox(this, ID_LISTBOX);
}

void TTestDialog::FillListBox()
{
  ListBox->AddString("Item 1");
}
```

コントロール通知メッセージに対する応答

親に知らせなくてはならないイベントが発生すると，コントロールはコントロール通知メッセージを親に送ります。ObjectWindowsでは，ダイアログが特定のコントロールからメッセージを受信すると，ダイアログオブジェクトがその応答を指定します。この指定は子IDメッセージ応答メンバ関数を定義することで行なわれます。このメンバ関数は，特定の子ウィンドウからの通知メッセージをダイアログが受信すると，自動的に呼び出されます。

次の宣言例で，*TTestDialog* は2つのコントロールに対する子ID応答メンバ関数を宣言しています。

```
class TTestDialog : public TDialog {
  :
void HandleButtonMsg(RTMessage Msg) =
  [ID_FIRST + ID_BUTTON];
void HandleListBoxMsg(RTMessage Msg) =
  [ID_FIRST + ID_LISTBOX];
};
```

各コントロール通知メッセージには，どの動作が行なわれたかを識別する通知コード（整数定数）がついています。たとえば，リストボックスの項目を選択すると，通知コード ***LBN_SELCHANGE*** のついたコントロール通知メッセージが送られます。ボタンをクリックすると，***BN_CLICKED*** コードが渡されます。なお，通知コードは *Msg.LParam* の上位ワードである *Msg.LP.Hi* で渡されます。

通常，子IDメッセージ応答メンバ関数で指定される応答は，次の例のように *Msg.LP.Hi* で渡された通知コードにより決まります。

```
void TTestDialog::HandleListBoxMsg(RTMessage Msg) {
  switch (Msg.LP.Hi)
  {
    case LBN_SELCHANGE :
      // 選択内容の変更を処理
      break;
    case LBN_DBLCLK :
      // 選択項目のダブルクリックを処理
      break;
  :
  }
}
```

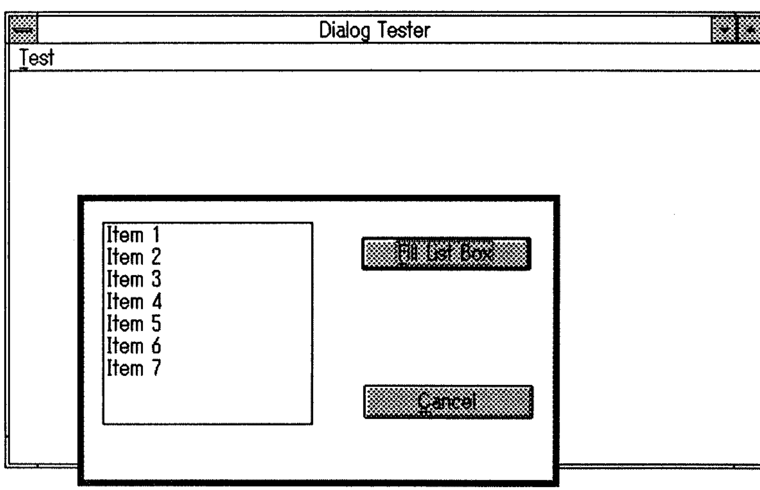
コントロール通知メッセージに対する応答を，ダイアログオブジェクトに定義する方法はすでに述べましたが，ダイアログのコントロールを通知メッセージに応答させるにはどうすればいいでしょうか。たとえば，ダイアログのボタンを押したときに何らかの動作をさせる場合は，コントロールクラスを定義して，そこで通知メンバ関数を定義する必要があります。さらに，ObjectWindowsのコントロールオブジェクトを，前の節で述べたように，ダイアログのコントロールに対応づける必要があります。

ダイアログとコントロールの通信例

次のプログラムで，*TTestDialog* は ***TDialog*** からの派生クラスです。*TTestDialog* では *HandleButtonMsg* と *HandleListBoxMsg* の2つを定義しています。この2つは，ダイアログのボタンコントロールとリストボックスコントロールからダイアログに送られる通知メッセージに応答します。*HandleButtonMsg* は，ボタンを押すと自動的に呼び出され，ダイアログのメンバ関数 *SendDlgItemMsg* を呼び出してリストボックスコントロールにテキスト項目を表示します。

TTestDialog は，モードつきダイアログとして，親ウィンドウのメンバ関数 ***ExecDialog*** で実行されます。

図 11.1
ダイアログアプリケーション



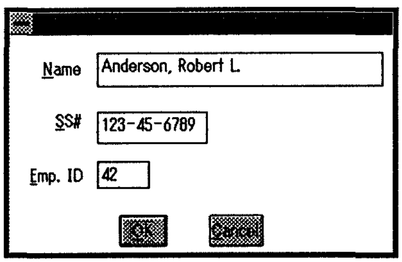
このDIALTEST.CPPプログラムは，EXAMPLESサブディレクトリにあります。

ダイアログ機能の拡張例

VDLGAPP.CPPが定義するダイアログは，ユーザにテキストの入力を要求し，入力された内容が正しいかどうかを調べるという複雑なものです。このプログラムが表示するダイアログボックスには，従業員名，社会保険番号，従業員IDを入力するためのエディットフィールドがついています。

ユーザがOKをクリックすると，***CanClose*** メンバ関数が，氏名，社会保険番号，従業員IDの入力に間違いがないことを確かめてからダイアログをクローズさせます。**図11.2** はこのダイアログです。

図11.2
ユーザ入力内容を確認するダイアログ



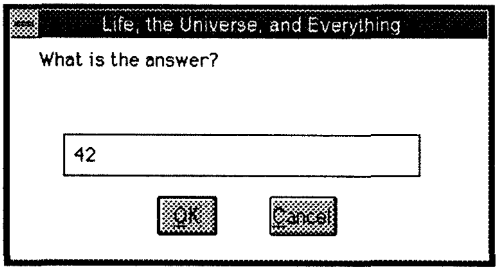
このプログラムはVDLGAPP.CPPという名前でEXAMPLESサブディレクトリにあります。

入力ダイアログ

入力ダイアログは，ObjectWindowsが提供するストックダイアログです。プロンプトを表示して，ユーザが入力した1行のテキストを取得します。このタスクはウィンドウを使うアプリケーションに共通なので，作成したアプリケーションで入力ダイアログを頻繁に使うはずです。

☞ 入力ダイアログを使うには，プログラムにINPUTDIA.Hを，プログラムの.RCファイルにINPUTDIA.DLGを含めます。

図 11.3
入力ダイアログ



入力ダイアログのコンストラクタは以下の5種類の引数をとります。

■ 親ウィンドウオブジェクトへのポインタ
■ ダイアログのタイトルとして表示するテキストへのポインタ
■ プロンプトとして表示するテキストへのポインタ
■ ユーザが入力したテキストを入れるテキストバッファへのポインタ
■ テキスト入力バッファのサイズ

以下は，入力ダイアログの構築例です。入力ダイアログのOKボタンを押すと，ユーザがダイアログのエディットコントロールに入力したテキストを*EditText*に入れます。

```
PTInputDialog AnInputDialog;
char EditText[79];

strcpy(EditText, "");
AnInputDialog = new TInputDialog(this, "Caption", "Prompt", EditText,
                                 sizeof(EditText));
```

入力ダイアログのコンストラクタに渡すテキストバッファは，入力ダイアログを初めて表示するときに表示させるデフォルトのテキスト文字列を保持するものです。したがって，テキストバッファの内容として，何らかの文字列を設定しておきます。ヌル文字列でもかまいません。

入力ダイアログのテキスト項目を指すポインタは，ダイアログが存続する間は有効にしておいてください。特に入力ダイアログをモードなしダイアログとして使う場合に重要です。ただし，実際には，次の例のように入力ダイアログはモードつきダイアログとして使う方が多いはずです。

```
void TSampleWindow::GetName()
{
  char EditText[79];

  strcpy(EditText, "Frank Borland");
  if (ExecDialog(new TInputDialog(this, "Data Entry", "Enter Name",
      EditText, sizeof(EditText))) == IDOK)
    MessageBox(HWindow, EditText, "Name is:", MB_OK);
  else MessageBox(HWindow, EditText, "Name is still:", MB_OK);
}
```

ファイルダイアログ

ファイルダイアログも ObjectWindows が ***TFileDialog*** クラスで提供するストックダイアログです。多くのアプリケーションで使われる File Open, Save, Save As などの関数で，ユーザにファイル名の入力を要求するときは必ずファイルダイアログを使います。**図 11.4** を参照してください。

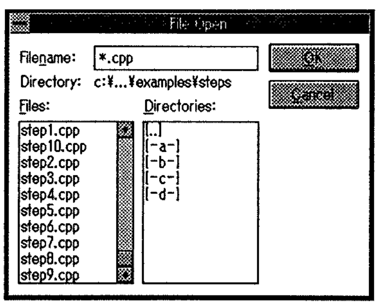


図 11.4
ファイルダイアログ

TFileDialog を使うには，プログラムに FILEDIAL.H を，プログラムの.RC ファイルに FILEDIAL.DLG を含める必要があります。

ファイルダイアログボックスのコンストラクタは，次の引数をとります。親ウィンドウを指すポインタ，リソース識別子，ファイル名またはファイルマスクの3つです。ファイルダイアログの用途がファイルのオープン用であればファイルマスク，ファイルの保管用であればファイル名になります。渡されるリソース識別子は，標準の Open ファイルダイアログボックスか標準の Save As ファイルダイアログボックスの識別子です。すなわち，***SD_FILEOPEN*** か ***SD_FILESAVE*** です。デフォルトのファイルマスクは，Open ファイルダイアログのファイル名引数で渡されます。この引数は Save As ファイルダイアログのデフォルトのファイル名も入ります。ファイル名引数は，Open ファイルダイアログと Save As ファイルダイアログの両方に対してユーザが選択したファイル名のリターンバッファとしても使われます。

次のコード例では，Open ファイルダイアログボックスが実行されます。"*.*" はファイルマスクとしてファイルダイアログのコンストラクタに渡されます。したがって，ファイルダイアログが最初に表示されたときに，カレントディレクトリ内のすべてのファイルがファイル名リストボックスに一覧表示されます。

```
char TempName[MAXPATH];
 :
_fstrcpy(TempName, "*.*");
if (GetApplication()->ExecDialog(new TFileDialog(this, SD_FILEOPEN,
                                    TempName)) == IDOK))
// TempNameの指定ファイルをオープン
 :
```

次のコード例では，Save As ファイルダイアログボックスが実行されます。ヌル文字列がデフォルトのファイル名としてファイルダイアログのコンストラクタに渡されます。したがって，ファイルダイアログを最初に表示するときに，デフォルトのファイルはいっさい選択されません。

```
char TempName[MAXPATH];
 :
TempName[0] = '¥0';
if (GetApplication()->ExecDialog(new TFileDialog(this, SD_FILESAVE,
                                    TempName)) == IDOK)
// 指定ファイルをTempNameにセーブ
 :
```

第12章

コントロールオブジェクト

ユーザ入力の転送を簡単に行なうためのユーザインターフェース要素は，まとめて**コントロール**と呼ばれます。Windowsアプリケーションのユーザにお馴染みの**チェックボックス**と**リストボックス**はそれぞれ別の種類のコントロールです。Windows側からみると，コントロールは単なる専用化されたウィンドウにすぎません。したがって，ObjectWindowsの***TControl***は***TWindow***から派生しています。

TControlから派生したクラスはWindowsのコントロールを表わすものです。アプリケーションでは，以下の表に示す派生クラスのインスタンスを使います。

表12.1
ObjectWindowsがサポートするWindowsコントロール

コントロール	クラス	用途
リストボックス	***TListBox***	ファイルなどの選択項目を一覧表示したスクロール可能なリスト。
スクロールバー	***TScrollBar***	スタンドアロンのスクロールバーで，ウィンドウやリストボックスのスクロールに使うものと外観が似ているもの。
プッシュボタン	***TButton***	テキストつきのプッシュボタン。
チェックボックス	***TCheckBox***	チェックマークをつけたり消したりできる箱型のボタンで，テキストがついているもの。
ラジオボタン	***TRadioButton***	選択するかしないかを選べるボタンで，通常相互に排他的なグループに使われる。
グループボックス	***TGroupBox***	左上隅にテキストがついたスタティックな矩形で，他のコントロールをグループにまとめるもの。
エディットコントロール	***TEdit***	ユーザがテキストを入力するためのフィールド。
スタティックコントロール	***TStatic***	ユーザが変更できないテキストフィールド。
コンボボックス	***TComboBox***	リストボックスとエディットコントロールを組み合わせたもの。

代表的なコントロールの画面上の外観を**図 12.1** に示します。

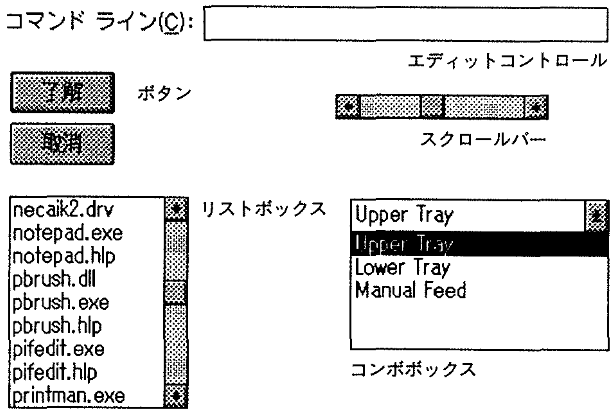


図 12.1
コントロールの例

ObjectWindows ではこれらの各種コントロールに対してすべてクラスが定義してあり，他のウィンドウ内で子ウィンドウとして利用できるようになっています。ダイアログボックスを構成するコントロールは，ダイアログリソースで指定されているコントロールインターフェース要素なので，対応する ObjectWindows オブジェクトは必要ありません（第 11 章『**ダイアログオブジェクト**』を参照)。この章では提供されているコントロールオブジェクトの機能を紹介し，ウィンドウ内でどのように使われるかを説明します。

コントロールオブジェクトの使用

コントロールは，通常親ウィンドウとともに作成され破棄されるので，両者の寿命は大体一致します。作成も破棄も ObjectWindows で自動的に処理されます。親ウィンドウの子リスト内にあるそれぞれの子ウィンドウは，親を作成するプロセスで作成され，親の破棄とともに破棄されます。

コントロールの構築と作成

コントロールオブジェクトは，ほぼ例外なく親ウィンドウのコンストラクタで構築しますが，構築する子の参照を親のデータメンバとして保持する必要はありません。子ウィンドウの参照は，親ウィンドウのメンバ関数 ***ChildWithID*** を呼び出せばいつでも得られます。ただし，***TControl*** からのすべての派生クラスのコンストラクタに渡す *AnID* 引数で各子ウィンドウ固有の識別子が指定されていることが必要です。プログラムで頻繁に使う子ウィンドウがある場合，その親のデータメンバとしてポインタを保持しておくと便利です。

コントロールオブジェクトのコンストラクタは，通常少くとも次の引数をとります。親ウィンドウオブジェクトを指すポインタ，コントロールの ID，左上隅の X 座標と Y 座標（原点位置は親のクライアント領域内），幅，高さの 6 つです。これらの引数を使って，コントロールオブジェクトの作成属性を設定し，***Attr*** 構造体に保持します。***TControl*** コンストラクタが *Attr.Style* に設定するスタイルには，***WS_CHILD***，***WS_VISIBLE***，***WS_TABSTOP***，***WS_GROUP*** があります。このデフォルトは，ObjectWindows の一部のコントロールクラスと同様に，下位のコントロールクラスで変更できます。ウィンドウ属性データメンバについては**表 10.1** を参照してください。

作成した各コントロールに対して，***Create*** または ***GetApplication( )->MakeWindow*** を直接呼び出さないでください。親ウィンドウに対して ***TWindowsObject*** のメンバ関数 ***SetupWindow*** を呼び出せば，自動的に作成されるからです。親ウィンドウの派生クラスで ***SetupWindow*** を定義し直せば，コントロールのセットアップ処理を行なうことができます。ただし，***SetupWindow*** を定義し直す場合は，基本クラスのメンバ関数 ***SetupWindow*** を必ず呼び出して，コントロールを作成します。

親ウィンドウにそのコントロールを指すポインタを保持させるようにデータメンバを定義することもできます。定義したら次に，親ウィンドウのコンストラクタでコントロールオブジェクトを構築します。最後に，***SetupWindow*** を定義し直してコントロールに必要なセットアップ処理を施します。残りの処理は ObjectWindows がしてくれます。

コントロールの破壊と削除

ObjectWindowsでは，コントロールウィンドウの破壊と削除も，親ウィンドウにまかされています。親ウィンドウが破壊され削除されると，コントロール要素が自動的に破壊され，対応するコントロールオブジェクトも自動的に削除されます。

コントロールとメッセージ処理

ウィンドウオブジェクトとコントロールオブジェクト間の通信は，ある意味でダイアログオブジェクトとコントロール要素間の通信に似ています。どちらの場合も，親ウィンドウがコントロールを操作して照会します。コントロールはコントロールイベントの通知メッセージを親に送ります。親は，これらの通知メッセージに対する子ID応答メンバ関数を定義できます。しかし，ダイアログのコントロールは，デフォルトではObjectWindowsのコントロールオブジェクトと対応していません。コントロールオブジェクトは常にウィンドウのコントロールと対応しています。この節では，ウィンドウとコントロール間の通信を説明します。ダイアログとコントロール間の通信については，第11章『**ダイアログオブジェクト**』を参照してください。

ウィンドウ内のコントロールの操作

Windowsはコントロールのメンバ関数を呼び出して，コントロールを操作し照会します。

ウィンドウのコントロールオブジェクトが親ウィンドウのオブジェクトのデータメンバと対応している場合は，コントロールのメンバ関数を呼び出すのは次のように簡単です。

```
ListBox->AddString("Scotts Valley");
```

コントロールオブジェクトのポインタがデータメンバとして保持されていない場合は，親のメンバ関数 ***ChildWithID*** を呼び出して，親の子リストから取得してください。次のコードは，親ウィンドウオブジェクトのメンバ関数の一部です。

```
PTListBox TheListBox;
  ⋮
TheListBox = (PListBox)ChildWithID(ID_LISTBOX);
TheListBox->AddString("Scotts Valley");
```

コントロール通知メッセージに対する応答

ウィンドウのコントロールに影響するイベントが発生すると，コントロールは**コントロール通知メッセージ**をウィンドウに送ります（第6章の節「**コントロールイベントへの応答**」を参照）。たいていの場合，コントロール通知メッセージは，次の例のように，子ID応答メンバ関数で親ウィンドウが処理します。

```
class TSampleWindow : public TWindow {
public:
  PTListBox ListBox;
  PTButton Button1, Button2;

  virtual void HandleListBoxMsg(RTMessage Msg) =
   [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
   [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
   [ID_FIRST + ID_BUTTON2];
}
```

応答メンバ関数の *Msg.LP.Hi* で ***LBN_SELCHANGE*** や ***BN_CLICKED*** などのコントロール通知コードが渡されます。次のように応答を記述して，重要な通知コードを処理するようにします。

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  switch (Msg.LP.Hi) {
    case LBN_SELCHANGE:
      // 選択内容の変更を処理
      break;

    case LBN_DBLCLK:
      // 選択項目のダブルクリックを処理
      break:
  }
}
```

コントロールオブジェクトをコントロール通知メッセージに応答させたい場合は，応答という動作をコントロールに組み込みます。たとえば，ユーザがクリックするとテキストが変化するボタンとか，数字しか入力できないエディットコントロールを設計する場合が考えられます。このような特殊な場合，親ウィンドウごとに目的の動作を複製するよりも，コントロールクラスで提供できると便利です。この点を考慮してスタンドアロンのクラスが用意されています。このクラスは多くのアプリケーションで何回も再利用できるものです。

コントロールを通知メッセージに応答させるには，次のように ***TControl*** からの派生クラスで**通知別**応答メンバ関数を定義します。

```
class TSpecializedListBox : public TListBox {
public:
  virtual void LBNSelChange(RTMessage Msg) =
    [NF_FIRST + LBN_SELCHANGE];
  virtual void LBNDblClick(RTMessage Msg) =
    [NF_FIRST + LBN_DBLCLICK];
}
```

通知別応答メンバ関数の関数宣言拡張子に，必ず ***NF_FIRST*** とコントロール通知コードの和を使ってください。

次に示すのは，***TSpecializedListBox*** :: ***LBNSelChange*** の定義です。

```
void TSpecializedListBox::LBNSelChange(RTMessage Msg)
{
  // 選択内容の変更を処理
}
```

コントロールと親ウィンドウを通知メッセージに応答させるには，通知別応答メンバ関数から，コントロールのメンバ関数 ***DefNotificationProc*** を呼び出してください。

コントロールフォーカスとキーボード

ダイアログボックス内でフォーカスを別のコントロールに移すとき，キーボードを使ったことがありますか。**TAB** を使えば，作成順にコントロールを巡回することができます。**矢印**キーを使えば，グループボックス内でフォーカスを別のラジオボタンに移し，選択を変えることができます。ObjectWindows ではこのキーボードインターフェースをエミュレートして，ウィンドウのコントロールに使えます。この「キーボード操作」機能を使いたければ，ウィンドウのメンバ関数 ***EnableKBHandler*** をコンストラクタで呼び出してください。

リストボックスコントロール

Windows プログラムのユーザにリストから項目を選択させるには，リストボックスを使うのが最も簡単です。プログラムの性質に応じて，ファイル，プリンタ，形状，結果の各種リストからの選択をユーザに要求できます。リストボックスは ***TListBox*** クラスによってカプセル化されています。***TListBox*** はリストボックスの作成，項目一覧の変更，項目一覧に関する問い合わせ，ユーザが選択した項目の識別の 4 つの用途に対しメンバ関数を定義しています。

リストボックスの構築と作成

TListBox には，親ウィンドウへのポインタ，ID，コントロールの *X*，*Y*，*W*，*H* の寸法を引数としてとるコンストラクタが 1 つあります。

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
```

TListBox のこのコンストラクタは ***TControl*** のコンストラクタを呼び出してから，***LBS_STANDARD*** を *Attr.Style* でリストボックスに対して指定されているデフォルトスタイルに追加します。***LBS_STANDARD*** は複数のスタイルの組み合わせであり，次の機能を持つリストボックスを指定します。

- リスト項目をアルファベット順にソートする（***LBS_SORT***）。
- リストボックスで発生したイベントを親に通知する（***LBS_NOTIFY***）。
- 境界線で囲まれている（***WS_BORDER***）。
- 垂直スクロールバーを持つ（***WS_VSCROLL***）。

リストボックスを構築したら，デフォルトのスタイルを保持している *Attr.Style* を変更することができます。たとえば，リストボックスの項目をソートしない場合は次のコードを使います。

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
ListBox->Attr.Style &= ~LBS_SORT;
```

TListBox からの派生クラスのコンストラクタでも *Attr.Style* を変更することができます。

すべての子ウィンドウの場合と同様，リストボックスのコントロール要素も，その親ウィンドウによって自動的に作成されます。

リストボックスの変更

リストボックスを作成したら，内容となる項目の登録が必要です。項目は必ず文字列かオーナー描画とします。後で項目の追加や削除，またはリスト全体の消去もできます。項目を登録したり追加するときは，リストボックスのメンバ関数 ***AddString*** を呼び出します。

```
ListBox->AddString("Item 1");
ListBox->AddString("Item 2");
ListBox->AddString("Item 3");
ListBox->AddString("Item 4");
ListBox->AddString("Item 5");
ListBox->AddString("Item 6");
```

AddString が失敗すると，負の値が返ります。

リストボックスは文字列の一覧を文字列の配列のように保持します。どちらもその要素をインデックスで管理するからです。*ListBox* が配列の場合，"Item 1" は *ListBox*[*0*] に，"Item 2" は *ListBox*[*1*] に，"Item 3" は *ListBox*[*2*] にというように格納されていきます。***AddString*** を呼び出すたびに，指定された文字列がデフォルトではアルファベット順に追加されます。スタイルに ***LBS_SORT*** がない場合，その文字列はリストの最後に追加されます。

リストボックスの内容がソートされる場合でもされない場合でも，次のように ***InsertString*** で指定の位置（インデックス）に新しい要素を挿入することができます。

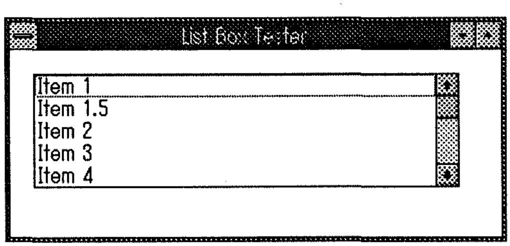
```
ListBox->InsertString("Item 1.5", 1);
```

この例では，インデックス 1 以降の位置に格納されているすべての文字列が 1 つずつ後ろに移動されます。挿入後のリストは次のようになります。

インデックス	項目
0	"Item 1"
1	"Item 1.5"
2	"Item 2"
3	"Item 3"
4	"Item 4"
5	"Item 5"
6	"Item 6"

表示結果を**図 12.2** に示します。

図 12.2
項目挿入後のリストボックス



新しい文字列を−1のインデックスに挿入すると，リストの最後に追加されます。

要素を削除するには ***DeleteString*** を呼び出します。次のようにメンバ関数を呼び出すとインデックス1の文字列("Item 1.5")が削除され，以降のインデックスにあるすべての文字列が1つずつ上に移動します。

```
ListBox->DeleteString(1);
```

リストボックスの文字列をすべて消去するには次のように ***ClearList*** を使います。

```
ListBox->ClearList();
```

リストボックスの照会

メンバ関数を呼び出すことによって，リストボックスオブジェクトで保持しているリスト情報を参照することができます。このようなメンバ関数は6つあります。

- ***GetCount*** はリスト内の項目数を返します。
- ***FindString*** と ***FindExactString*** はリストボックスを検索し，指定された文字列を探し出します。指定位置のインデックスから開始し，必要であればリストの先頭に戻って検索します。***FindString*** は，指定された文字列を**含む**文字列をリストボックスから検索します。一方，***FindExactString*** は指定された文字列で**始まる**文字列を検索します。
- ***GetString*** は，整数引数によって指定されたインデックス位置にある文字列をリストから取得します。***LPSTR*** 引数は，文字列を格納するバッファを指します。***GetString*** は文字列の長さを示す整数も返します。
- ***GetStringLen*** は，指定位置の文字列の長さだけを返し，文字列自体は返しません。
- ***GetSelIndex*** は，現在選択されている文字列のインデックスを返します。単項目選択のリストボックスのみに使います。

リストボックスからの選択項目の取得

リストボックスでユーザができる操作は3つあります。リストのスクロール，項目のクリックとダブルクリックです。リストボックスをユーザが操作すると，Windowsは**リストボックス通知**メッセージをリストボックスの親ウィンドウに送ります。

リストボックス通知（***LBN***）メッセージは*Msg.LP.Hi*のリストボックス通知コード（正定数）で操作の性質を特定します。代表的な***LBN***コードを次の表に示します。

表 12.2
リストボックス通知メッセージ

WParam	操作
LBN_SELCHANGE	1回のマウスクリックでオプションが選択された。
LBN_DBLCLK	ダブルクリックでオプションが選択された。
LBN_SETFOCUS	項目のクリックかダブルクリック，あるいは**TAB**キーでリストボックスにフォーカスが当てられた。このメッセージは***LBN_SELCHANGE***と***LBN_DBLCLK***に先行します。

親ウィンドウは，通常，定義した子ID応答メンバ関数で，リストボックスからの通知メッセージを処理します。リストボックスからの通知メッセージを処理する親ウィンドウのメンバ関数の例を以下に示します。

LBOXTEST.CPPの一部です。

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  int Idx;
  char Str[10];
  if (Msg.LP.Hi == LBN_SELCHANGE)
  {
    Idx = ListBox->GetSelIndex();
    if (ListBox->GetStringLen(Idx) <= sizeof(Str))
    {
      ListBox->GetSelString(Str, sizeof(Str));
      MessageBox(HWindow, Str, "You selected:", MB_OK);
    }
  }
}
```

ユーザが選択を行なうと，*LParam* の上位ワードに ***LBN_SELCHANGE*** 通知コードが入ります。この場合，選択された文字列の長さを取得し，*Str* バッファに収まるかどうかを調べてから文字列を返し，メッセージボックスに表示します（**図 12.3** を参照）。

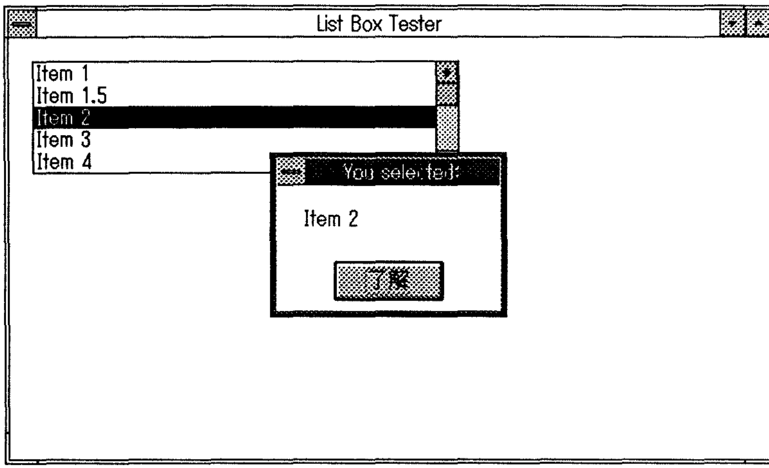


図 12.3
ユーザがリストボックス項目を選択したときの応答

選択された項目は***GetSelString***, ***GetSelIndex***, ***SetSelString***, ***SetSelIndex*** の 4 つの ***TListBox*** メンバ関数によって処理されます。***Get***... メンバ関数は選択された文字列とそのインデックスを取得します。***Set***... メンバ関数はユーザによる指示を無視し，文字列やインデックスを指定して特定の項目を強制的に選択させます。このため，項目が指定されていなくても自動的に表示されます。

LBoxTest は，リストボックスつきのウィンドウを作成するプログラムです。アプリケーションが起動すると，リストボックスがメインウィンドウに表示されます。ユーザがリストボックスの項目を選択すると，このリスト項目を表示したダイアログが現われます。メインウィンドウはリストボックスコントロールへの参照を *ListBox* データメンバに保持しているので，コントロールを簡単に操作することができます。

このファイルは LBOXTEST.CPP という名前で EXAMPLES サブディレクトリに収めてあります。

コンボボックス

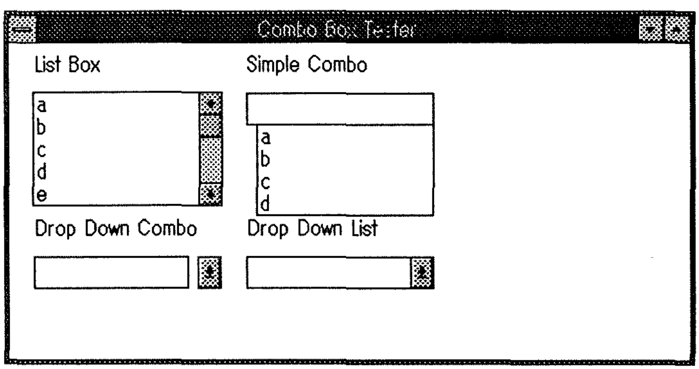
コンボボックスコントロールは，リストボックスとエディットコントロールの2つのコントロールを組み合わせたものです。コンボボックスではユーザからの入力をリストボックスの選択項目として，またはエディットコントロールのテキストとして取得します。

TComboBox は ***TListBox*** から派生しており，リスト項目の変更，照会，選択に用いるメンバ関数を継承しています。***TComboBox*** はコンボボックスのリスト部分を操作するためのメンバ関数も提供しているので，必要に応じてリストを**ドロップダウン**することもできます。

3 種類のコンボボックス

コンボボックスには，**シンプル**，**ドロップダウン**，**ドロップダウンリスト**の3種類があります。これら3種類のコンボボックスとリストボックスの外観を**図12.4**に示します。

図 12.4
3 種類のコンボボックスとリストボックス



シンプルコンボボックス

コンボボックスは，どれでも常時エディット領域を表示しています。また，コンボボックスによっては，カメが甲羅に首を引っ込めるようにリストボックス領域を出したり隠したりできるものがあります。シンプルコンボボックスはリストボックス領域を隠すことはできず，いつでも表示しています。シンプルコンボボックスにはエディットコントロールがあり，ユーザはここにテキストを入力して編集できます。入力したテキストがリスト内の項目のどれかと一致すると，その項目が選択されます。

ドロップダウンコンボボックス

ドロップダウンコンボボックスはシンプルコンボボックスに似ていますが，初期状態ではリスト領域は表示されません。ユーザがエディットコントロールの右にある下向き矢印をクリックすると，リスト領域が表示されます。選択が済むと，リストボックスは隠れます。多数のコントロールを狭い領域に配置したい場合には，ドロップダウンコンボボックスやドロップダウンリストコンボボックスが便利です。使わないときの占有スペースが，シンプルコンボボックスやリストボックスよりもはるかに小さくてすむからです。

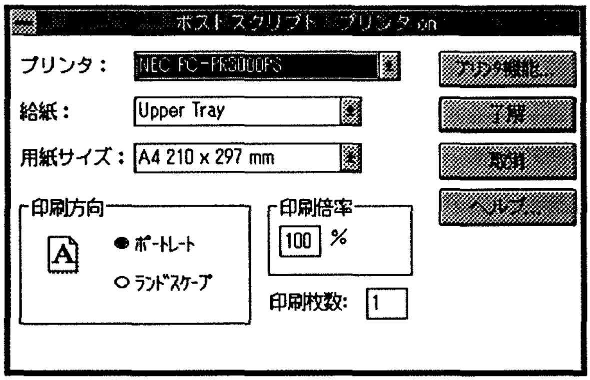
ドロップダウンリストコンボボックス

ドロップダウンリストコンボボックスのリストボックスは，ドロップダウンコンボボックスのリストボックスに似た動作をします。つまり，必要なときに現われ，必要でないときには隠れています。この2つのコンボボックスの違いは，エディット領域の動作です。ドロップダウンコンボボックスのエディット領域は通常のエディットコントロールの働きをしますが，ドロップダウンリストのエディット領域はリスト項目のテキストを表示するだけで，ユーザが直接変更することはできません。

コンボボックスの選択

ドロップダウンリストコンボボックスが便利なのは，リスト領域に一覧表示された項目だけを選択対象にする場合です。たとえば，プリンタを選択する場合，システムからアクセスできるプリンタだけしか選択できません（**図 12.5** 参照）。

図 12.5
ドロップダウンリストコンボボックス



他方，ドロップダウンコンボボックスは，リスト項目以外の入力も受け付けます。たとえば，オープンまたはセーブするディスクファイルを選択するときに使えます。ユーザはディレクトリを調べてリスト内の該当ファイルを探し出すことも，フルパス名やファイル名をエディット領域にタイプすることもできます。リスト領域に出ていないファイル名もタイプできます。

コンボボックスの構築

TComboBox のコンストラクタには，親ウィンドウへのポインタ，コントロールID，位置とサイズのデータという通常のコントロールコンストラクタ用の引数に加えて，*AStyle* と *ATextLen* も使えます。

標準の Windows コンボボックススタイルの ***CBS_SIMPLE***，***CBS_DROPDOWN***，***CBS_DROPDOWNLIST*** のいずれかが，コンボボックスのコンストラクタに引数 *AStyle* として渡されます。コンボボックスのデータメンバ *Attr.Style* の設定には，次に示すように，引数 *AStyle* の他にデフォルトの付加スタイルも使います。

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP |
             CBS_SORT | CBS_AUTOHSCROLL | WS_VSCROLL | AStyle;
```

次のコード例では，コンボボックスのデータメンバ *Attr.Style* を変更して，コンボボックスのリストボックスをソートしないようにしています。

```
Combo3 = new TComboBox(this, ID_COMBO3, 190, 160, 150, 100,
                       CBS_DROPDOWNLIST, 40);
Combo3->Attr.Style &= ~CBS_SORT;
```

引数 *ATextLen* は，コンボボックスのエディットコントロール部分の最大長です。

コンボボックスの変更

TComboBox には，2つのメンバ関数 ***ShowList*** と ***HideList*** が定義されています。***ShowList*** はドロップダウンコンボボックスとドロップダウンリストコンボボックスのリスト領域を表示し，***HideList*** は隠します。ともに手続きなので引数はとりません。また，ユーザがエディット領域の右にある下向き矢印をクリックしたとき，これらのメンバ関数を呼び出してリストを表示させたり隠したりする必要はありません。表示または非表示の処理は，Windows コンボボックスの自動機構になっているからです。この2つのメンバ関数は，リストの表示，非表示を強制的に行なうときに便利です。

アプリケーション例：CBoxTest

プログラム *CBoxTest* は，**図 12.4** に示したアプリケーションを作成します。3種類のコンボボックスをすべて使っています。*Combo 1* はシンプルコンボボックス，*Combo 2* はドロップダウンコンボボックス，*Combo 3* はドロップダウンリストコンボボックスです。

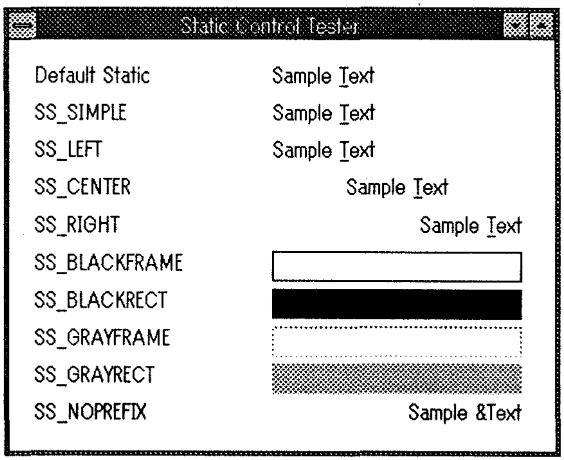
Show ボタンをクリックすると，右下のコンボボックス *Combo 3* のリストが強制的に表示され，Hide ボタンをクリックすると強制的に非表示になります。Show ボタンのときはメンバ関数 ***ShowList***，Hide ボタンのときは ***HideList*** が呼び出されます。

このファイルはCBOXTEST.CPPという名前でEXAMPLESサブディレクトリに収めてあります。

スタティックコントロール

一般に，スタティックコントロールは変更できないテキストや簡単なグラフィックスであり，ウィンドウやダイアログに表示できるものです。ユーザがスタティックコントロールと対話することはありませんが，プログラムでテキストを変更することはできます。種々のスタティックコントロールのスタイルと対応するWindowsスタイル定数を**図12.6**に示します。

図12.6
スタティックコントロール



スタティックコントロールの構築

スタティックコントロールは通知メッセージを送ることもなければ，IDによるプログラム操作もほとんどないので，固有のIDを設定する必要がありません。スタティックコントロールには，通例としてコントロールの“don't care”IDである-1を使います。

TStaticのコンストラクタは，コントロールに使用する通常の引数の他に，テキスト長を指定する引数もとります。この引数で，***TStatic***オブジェクトが**転送**するテキストの長さが渡されます。データ転送の説明と***TStatic***のデータメンバ***TextLen***の詳細については，この章の最後の節「**コントロールデータの転送**」を参照してください。

TStaticコンストラクタは，***TControl***のコンストラクタを呼び出してから，データメンバ*Attr.Style*に***SS_LEFT***（テキストの左詰め）スタイルを追加し，***WS_TABSTOP***スタイルを除去します。スタティックコントロール用に設定されているデフォルトのスタイルを変更するには，次のように*Attr.Style*を変更します。

```
AStatic = new TStatic(this, -1, "&Text", 20, 50, 200, 24, 6);
AStatic->Attr.Style = AStatic->Attr.Style & ~SS_LEFT | SS_CENTER;
```

構築したスタティクコントロールのスタイルを変更する必要がなければ，その参照をローカル変数として保持する必要もありません。スタティックコントロールをいったん作成したら操作することはほとんどないので，その参照を親ウィンドウのデータメンバとして保持する必要はありません。

スタティックコントロールに使えるオプションとして，テキスト文字列に含まれる1文字あるいは文字群に下線をつけることができます。この実現方法と効果は，メニュー項目の最初の文字に下線をつけるのに似ています。つまり，下線をつける文字の前に&文字を挿入するだけです。たとえば，'Text'のTに下線をつけるには，コンストラクタ呼び出しで文字列'&Text'を渡します。文字'&'を文字列に入れたい場合は，Windowsのスタティックスタイル***SS_NOPREFIX***を使います（**図12.6**参照）。

スタティックコントロールの照会

GetText メンバ関数を使うと，スタティックコントロールが現在保持しているテキストを問い合わせることができます。

スタティックコントロールの変更

TStatic には，スタティックコントロールのテキストを変更するためのメンバ関数が2つあります。***SetText*** は，スタティックのテキストを渡された ***LPSTR*** 引数に設定します。***Clear*** はスタティックのテキストを消去します。ただし，スタイル ***SS_SIMPLE*** で作成したスタティックコントロールのテキストは変更できません。

アプリケーション例：StatTest

STATTEST.CPP アプリケーションは，**図 12.6** に示すスタティックコントロールをテストするアプリケーションを作成します。"Default Static" とか "SS_SIMPLE" といったラベルも，"Sample Text" とか黒い箱や灰色の箱と同様にスタティックコントロールです。

このファイルは STATTEST.CPP という名前で EXAMPLES サブディレクトリに収めてあります。

エディットコントロール

エディットコントロールは矩形のコントロールで，境界枠がある場合とない場合があります。ユーザからの入力テキストを取得するのに使います。エディットコントロールはユーザがテキストを**入力する**ことも，そのテキストをプログラム上で変更することもできます。Cut，Copy，Paste など，エディットコントロールに対する多くの操作により，クリップボードとのテキストのやり取りが行なわれます。次の操作を行なえます。

- ユーザによるテキスト入力
- アプリケーションによるテキストの動的表示
- クリップボードへの貼り付け，複写，切り抜き
- 複数行の編集（テキストエディタに便利）

Windowsには，多種類のユーザイベントに応答できるエディットコントロールが備わっています。例えば，エディットコントロールの単語をダブルクリックすると，その語が選択されます。***DEL*** を押すと，カーソル位置の文字が削除されます。このようなWindowsのエディットコントロールのテキスト編集機能を，ObjectWindowsはいっそう拡充しています。

2つのエディットコントロールを持つウィンドウを**図12.7**に示します。

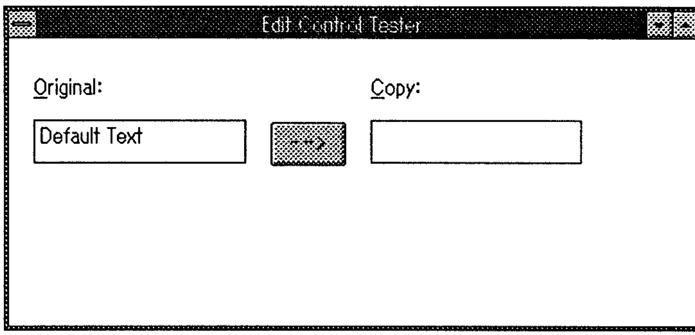


図12.7
エディットコントロールつきのウィンドウ

エディットコントロールの構築

TEdit のコンストラクタでは，通常のコントロールコンストラクタ引数である親ウィンドウを指すポインタ，コントロールID，位置とサイズのデータを使います。この他に，論理引数の *Multiline* と ***LPSTR*** 型の引数 *AText* を使います。*Multiline* で，複数行のエディットコントロールを作成するかどうかを指示します。別の引数 *ATextLen* には，エディットコントロールに入力できる文字の最大数が入ります。テキストはヌルで終わる文字列でなければならないので，エディットコントロールに実際に入力できる文字数は，渡される引数 *ATextLen* よりも1文字分少なくなります。

次のコード例では，単一行と複数行のエディットコントロールが構築されます。どちらのエディットコントロールにも，ユーザは39文字まで入力することができます。

```
SingleLineEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
MultilineEdit = new TEdit(this, ID_MEDIT, "Default Text",
                          20, 50, 150, 30, 40, TRUE);
```

WS_CHILD，***WS_VISIBLE***，***WS_TABSTOP***，***ES_LEFT***，***ES_AUTOHSCROLL***，***WS_BORDER*** は，すべてのエディットコントロールに設定されるデフォルトのスタイルです。*Multiline* が真の場合，そのコントロールはさらに別のデフォルトスタイル ***ES_MULTILINE***，***ES_AUTOVSCROLL***，***WS_HSCROLL***，***WS_VSCROLL*** が付加された複数行のエディットコントロールになります。

エディットコントロールを構築した後でも，次のようにしてスタイルを変えることができます。

```
BorderlessEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
BorderlessEdit->Attr.Style &= ~WS_BORDER;
```

エディットコントロールのメンバ関数 ***Create*** が呼び出されると，上のコードでは図 **12.8** のエディットコントロールが作成されます。

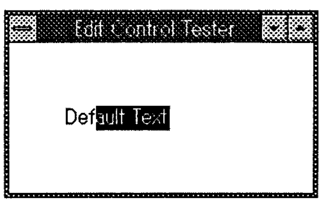


図 12.8
枠なしのエディットコントロール

エディットコントロールの一般的なスタイルを次の表に示します。

表 12.3
エディットコントロールの一般的なスタイル

スタイル	表示
ES_LEFT	左詰め
ES_CENTER	中央揃え
ES_RIGHT	右詰め
ES_MULTILINE	複数行のエディットコントロール
ES_UPPERCASE	大文字に変換
ES_LOWERCASE	小文字に変換
ES_AUTOVSCROLL	複数行エディットコントロールを自動的に垂直方向スクロール
ES_AUTOHSCROLL	自動水平方向スクロール
WS_BORDER	エディットコントロールの周囲に矩形枠を追加
WS_VSCROLL	垂直スクロールバーを追加
WS_HSCROLL	水平スクロールバーを追加

クリップボードと編集操作

次の ***TEdit*** メンバ関数を使って，エディットコントロールと Windows クリップボードの間でテキストを直接転送することができます。

- ***Cut***
- ***Copy***
- ***Paste***

次の編集メンバ関数も利用できます。

- ***Clear***（***TStatic*** から派生）
- ***DeleteSelection***
- ***Undo***

さらに，もう1つ利用できる編集メンバ関数として，論理メンバ関数の ***CanUndo*** があります。これは，最後に行なった変更を取り消しできるかどうかを決めるものです。

これらのメンバ関数に，編集メニュー経由でアクセスさせる場合も数多くあります。エディットコントロールは，Edit | Copy や Edit | Undo のようなメニュー選択に自動的に応答します。***TEdit*** では，***CMEditCopy*** や ***CMEditUndo*** のようなコマンド別のメンバ関数を定義します。これらのメンバ関数は，エディットコントロールの親ウィンドウにある特定のメニュー項目（コマンド）を選択すると呼び出されます。***CMEditCopy*** は ***Copy*** を呼び出し，***CMEditUndo*** は ***Undo*** を呼び出します。

RCOWL.H に定義されている特定のメニュー ID を持つメニュー項目を選択したときに呼び出されるメンバ関数を次の表に示します。

表 12.4
メニュー ID と呼び出されるメンバ関数

メニュー ID	呼び出されるメンバ関数
CM_EDITCUT	***Cut***
CM_EDITCOPY	***Copy***
CM_EDITPASTE	***Paste***
CM_EDITCLEAR	***Clear***
CM_EDITDELETE	***DeleteSelection***
CM_EDITUNDO	***Undo***

これらのメニュー ID のついた項目の編集メニューを追加するだけでよく，メンバ関数を記述する必要はありません。

エディットコントロールの照会

テキスト入力を確認したり，入力内容を後で使うために格納したり，または入力内容を別のコントロールに複写したりするために，エディットコントロールに問い合わせる場合がよくあります。***TEdit*** は，このような場合の照会用メンバ関数をいくつか提供しています。エディットコントロールの照会および変更用メンバ関数の多くは，行番号または行中の文字の位置を返すか，あるいはプログラマに指定を要求します。これらのインデックスはすべてゼロから始まります。すなわち，第 1 行は行 0 で，行中の第 1 文字目は文字 0 ということになります。照会用のメンバ関数の中でも ***GetText***，***GetLine***，***GetNumLines***，***GetLineLength*** が最も重要です。

エディットコントロールからテキストを返す ***TEdit*** の照会メンバ関数は，テキストのフォーマット情報を保持していることに気づかれるはずです。これは，複数行にテキストを表示できる複数行エディットコントロールを使う場合以外は関係ありません。エディットコントロールの複数行にまたがるテキストが返されると，各行末には**復帰**（0ｘ0D）と**改行**（0ｘ0A）の2文字が追加されます。復帰と改行が追加されたテキストをエディットコントロールに挿入したり，クリップボードから貼り付けたり，ファイルに書き込んだり，プリンタに出力したりすると，エディットコントロール内と同様に行送りが生じます。したがって，一定の文字数を取得する照会メンバ関数を使う場合には，行送りに2文字余計に必要になることを考慮してください。

GetText は，エディットコントロール内のテキストを取得します。そして，***LPSTR*** 引数で指示された文字列バッファにエディットコントロールの内容を書き込みます。行送りも含めて第2引数で指定した文字数に達するまで書き込まれます。呼び出しの値としては，コピーした文字数を返します。エディットコントロールが空であれば，0が返されます。

次の手続きはエディットコントロールからテキストを取得し，これを *TextBuffer* 引数で返すものです。

```
void TTestWindow::ReturnText(LPSTR TextBuffer, int BufferSize)
{
  char TheText[20];
  if (BufferSize >= sizeof(TheText) &&
      (Edit->GetText(TheText, sizeof(TheText) !=0))
    strcpy(TextBuffer, TheText);
  else strcpy(TextBuffer, "")
}
```

GetLine は ***GetText*** よりもさらに専用化したメンバ関数です。複数行エディットコントロールで使うと，整数引数で指定された行のテキストを返します。行0が最初の行になります。

GetNumLines は，複数行エディットコントロールに入力された行数を返します。***GetLine*** で行のテキストを取得する前に，エディットコントロールに何行入力されているかを ***GetNumLines*** を使って調べておいてください。

GetLineLength は，複数行エディットコントロール内の指定行の文字数を返します。指定された行はあるが，その行に何も文字がない場合には0が返されます。***GetLineLength***，***GetNumLines***，***GetLine*** を一緒に使って，複数行エディットコントロールからうまくテキストを取得してください。

```
void TTestWindow::ReturnLine(int LineNum, LPSTR TextBuffer,
                                int BufferSize) {
  char TheText[20];
  if ((Edit->GetNumLines >= LineNum) &&
      (Edit->GetLineLength(LineNum) >= sizeof(TheText)) &&
      (Edit->GetLine(TheText, sizeof(TheText)) != 0))
    strcpy(TextBuffer, TheText);
  else
    strcpy(TextBuffer,"")
}
```

GetText よりさらに専用化したメンバ関数として ***GetSubText*** もあります。引数として ***LPSTR*** と2つの整数をとります。これらの整数は，エディットコントロールのテキストの先頭と末尾のインデックスを示すもので，テキストの範囲を表わします。最初の文字はインデックス0の位置にあります。複数行エディットコントロールでは，インデックスは最初の行から順に数えられ，行送りは2文字とみなされます。この方法でエディットコントロールのテキストがプログラムで再フォーマットされ，表示，プリント，他のエディットコントロールへの転送，クリップボードへのコピーが行なわれます。

メンバ関数 ***GetSubText*** を呼び出すときに引数として渡すインデックスは，どうやって取得しますか。1つには，***GetSelection*** を使う方法があります。これは，現在選択中かハイライト表示中のテキストの先頭と末尾のインデックスを返すメンバ関数です。普通はユーザがテキストを選択しますが，***SetSelection*** を使えばプログラムでも選択できます(「**エディットコントロールの変更**」の節を参照)。***GetSelection*** が返す末尾のインデックスは，最後に選択された文字の次の位置を指します。

エディットコントロールからのテキストを処理するアプリケーションで処理時間を短縮することができます。論理関数であるメンバ関数 ***IsModified*** は，エディットコントロールのテキストに変更が加わった場合に真を返し，変更がない場合には偽を返します。偽が返されるので，***GetText*** や ***GetLine*** を呼び出す必要がありません。変更フラグを偽に設定し直すには ***ClearModify*** を使います。

GetLineIndex と ***GetLineFromPos*** の2つのメンバ関数を使うと，複数行エディットコントロール内のテキストの位置が得られます。***GetLineIndex*** は，整数引数で指定した行に先行するすべての行に含まれる文字数（行送りのための2文字を含む）を返します。指定した行がない場合にはすべての文字が返されます。***GetLineFromPos*** は，文字位置を示すインデックスを取得し，その位置に対応する行番号を返します。この2つのメンバ関数は，複数行エディットコントロールの内容と行構造を照会する場合に便利です。

エディットコントロールの変更

これまでのデータ入力プログラムでは，プログラムで直接エディットコントロールを変更する必要はなかったはずです。ユーザがテキストを変更すると，プログラムが照会用のメンバ関数を使ってテキストを読み込んでいました。しかし，エディットコントロールの使い方が多様化すると，アプリケーションにテキストの置換，挿入，消去，選択を明示的に行なわせたい場合が出てきます。ObjectWindows アプリケーションでは，***TEdit*** クラスのインスタンスのメンバ関数を呼び出して，プログラム上でエディットコントロールのテキストを操作することができます。

テキストの削除

Clear は，エディットコントロール内のテキストをすべて削除します。このメンバ関数はウィンドウ内のエディットコントロールの一部または全体を削除するリセット動作の一部として使うことができます。***DeleteSelection*** は，エディットコントロールで現在選択されているテキストを削除します。完全なテキストエディタでは，メニュー選択で ***Clear*** と ***DeleteSelection*** を呼び出せる場合があります(206 ページの「**クリップボードと編集操作**」を参照)。***DeleteSelection*** は論理関数であり，現在選択されているテキストがない場合は偽を返します。

DeleteSubText は ***DeleteSelection*** に似ていますが，渡された先頭位置と最終位置の間にあるテキストを削除する点が異なります。除去されるテキストには先頭位置の文字は含まれますが，最終位置の文字は含まれません。

DeleteLine は，指定した行のテキストをすべて削除します。ただし，行送りのための 2 文字は消去しないので，他の行には影響が及びません。

テキストの挿入

Insert は ***Paste*** に似ていますが，クリップボードからではなく渡された引数からテキストを取得します。選択されている既存のテキストを削除しますが，挿入したテキストをハイライトしません。***Insert*** を使うことによって，テキスト専用クリップボードの実現をプログラムで管理することができます。

SetText の機能は，***Clear*** と ***Insert*** を組み合せたものです。エディットコントロールの内容をすべて削除し，渡された引数のテキストを挿入します。たとえば，主にカリフォルニア州から注文を受ける受注システム用に受注画面を設定し直すには，次のようにして ***SetText*** を呼び出し，入力フィールド *state*（州）を設定し直します。

```
StateField->SetText("CA");
```

テキスト選択とスクロールの強制

SetSelection は，渡された先頭位置と最終位置の間にあるテキストを強制的に選択したり，ハイライト表示にします。最終位置の文字は含まれません。

Scroll は，複数行エディットコントロール内でのスクロールを強制します。***Scroll*** は次の2つの整数引数をとります。水平にスクロールする文字数を指定する引数と垂直にスクロールする行数を指定する引数です。正の整数で右または下にスクロールし，負の整数で左または上にスクロールします。複数行エディットコントロールだけが垂直にスクロールできます。

プログラム例：EditTest

EditTest は，2つのエディットコントロール，2つのスタティックコントロール，1つのボタンのついた，親ウィンドウとして使われるメインウィンドウを表示させるプログラムです。このウィンドウは204ページの**図12.7**に示したものです。

ユーザがボタンをクリックすると，左のエディットコントロール（*Edit 1*）にあるテキストが右のエディットコントロール（*Edit 2*）に複写されます。データメンバ *Attr.Style* でデフォルトの作成スタイルとして ***ES_UPPERCASE*** が追加されているので，*Edit 2* に複写されたテキストは大文字に変換されます。*Edit 1* でテキストを選択しないと，*Edit 1* のテキスト全部が *Edit 2* に複写されます。*Edit 1* でテキストを選択した場合は，選択部分だけが複写されます。

どちらのエディットコントロールでも入力フォーカスを持つ方で，**Edit** メニューの機能が使えます。

このファイルはEDITTEST.CPPという名前でEXAMPLESサブディレクトリに収めてあります。

プッシュボタンコントロール

プッシュボタンはコマンドボタンとも呼ばれ，ボタンが押されるたびに決められた作業を実行するものです。プッシュボタンのスタイルには2つあります。***BS_PUSHBUTTON*** と ***BS_DEFPUSHBUTTON*** で，どちらも ***TButton*** から派生しているので ***TButton*** のインスタンスとして使えます。デフォルトのプッシュボタンには太い黒枠がついていて，通常デフォルトのユーザ応答を示します。

TButton のコンストラクタには，コントロールコンストラクタの通常の引数である親ウィンドウへのポインタ，コントロールID，位置とサイズのデータが使えます。さらに，論理引数 *IsDefaultButton* と ***LPSTR*** 型引数 *Text* も使えます。*IsDaultButton* は，ボタンがデフォルトのプッシュボタンかどうかを示します。*Text* には，ボタンに表示するテキストが入ります。たとえば，ボタンが属する親のコンストラクタからの次の文では，「Test Button」というタイトルつきのボタンが構築されます。

```
Push1 = new TButton(this, ID_BUTTTON, "Test Button",
                    38, 48, 316, 24, FALSE);
```

ボタンメッセージへの応答

ユーザがプッシュボタンをクリックするたびに，ボタンの親ウィンドウはボタンから通知メッセージを受信します。これらのメッセージに親ウィンドウを応答させるには，子ID応答メンバ関数を定義します。

次の例では，識別子 ***ID_BUTTON*** を持つボタンからの通知メッセージへの応答として，***TTestWindow*** :: ***HandleButtonMsg*** がメッセージボックスを表示させます。

```
class TTestWindow : public TWindow {
public:
  PTRadioButton Button;
  virtual void HandleButtonMsg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON];
void TTestWindow::HandleButtonMsg(RTMessage Msg)
{
  MessageBox(HWindow, "clicked", "The button was:", MB_OK);
}
```

プッシュボタンに対してWindowsで定義されている通知コードは***BN_CLICKED***だけなので，これをチェックする必要はありません。

チェックボックスとラジオボタン

チェックボックスとラジオボタンをまとめて，**選択ボックス**と呼ぶことがあります。どちらもユーザからの入力を選択という形で受け取るためです。チェックボックスもラジオボタンも通常は，**チェック状態**か**非チェック状態**のどちらかの状態になっています。特殊なチェックボックスには，**淡色表示**を含めて3つの状態があります。チェックボックスが**淡色表示**のときは選択できない状態なので，ボックスをチェックすることはできません。

通常，チェックボックスはユーザの指定どおりに（たいていは複数の）オプションを取得するのに使い，ラジオボタンは相互に排他的なオプション群から1つを取得するのに使います。たとえば，チェックボックスでユーザはロードするフォントを複数選択できますが，ラジオボタンは特定の文字に使う書体を指定する場合などに使います。

チェックボックスもラジオボタンもObjectWindowsのクラス***TCheckBox***と***TRadioButton***のインスタンスとして表わされます。***TCheckBox***は***TButton***から，***TRadioButton***は***TCheckBox***から派生しています。

チェックボックスとラジオボタンの構築

TCheckBoxと***TRadioButton***のコンストラクタには，親ウィンドウへのポインタ，コントロールID，位置データ，サイズデータという通常のコントロールコンストラクタの引数を使えます。さらに，***LPSTR***型の***Text***引数と***PTGroupBox***型の*AGroup*引数も使えます。*Text*には，デフォルトで，選択ボックスの右側に表示するテキストが入ります。*AGroup*は，選択ボックスを論理グループにまとめるのに使える***TGroupBox***オブジェクトを指すポインタです。*AGroup*をNULLにすると，その選択ボックスは論理グループに属さないことになります。(216ページの「**グループボックス**」を参照)。

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box",
                         38, 102, 176, 108);
CheckBox = new TCheckBox(this, ID_CHECKBOX, "Check Box Test",
                         235, 12, 150, 26, GroupBox);
```

TCheckBox と ***TRadioButton*** が設定するデフォルトのスタイルは，***BS_AUTOCHECKBOX*** と ***BS_AUTORADIOBUTTON*** です。これらのスタイルでは，選択ボックスをクリックすると，Windows が自動的にチェック状態と非チェック状態を交互に切り換えます。「自動」スタイルのラジオボタンをクリックすると，同一グループ内の他のボタンは自動的にチェックマークが消されます。次のように，*Attr.Style* を設定し直して，「非自動」の ***BS_CHECKBOX*** や ***BS_RADIOBUTTON*** スタイルを指定すると，選択ボックスのチェック状態はプログラムで管理することになります。

```
CheckBox->Attr.Style |= BS_LEFTTEXT;
```

選択ボックス状態の照会

選択ボックスの状態を問い合わせるとその状態がわかり，これに対処することができます。ラジオボタンとチェックボックスには，チェックされているかいないかの 2 つの状態があります。選択ボックスの状態の照会には，***TCheckBox*** の ***GetCheck*** メンバ関数を次のように使います。

```
MyState = CheckBox->GetCheck();
```

GetCheck の戻り値を，定義済みの定数 ***BF_UNCHECKED, BF_CHECKED, BF_GRAYED*** と比較すると，ボックスの状態を判定できます。

選択ボックス状態の変更

チェックマークをつけたり消したりすることによって選択ボックスの状態を変更するのは，プログラマではなくユーザの仕事のように思われます。しかし，選択ボックスの状態をプログラムで直接制御しなければならない場合も出てきます。たとえば，以前に選択されセーブされているオプションを表示する場合があります。チェックボックスの状態を変更するために，***TCheckBox*** 型では 4 つのメンバ関数が定義されています。すなわち，***Check, Uncheck, Toggle, SetCheck*** で，***SetCheck*** が最も一般的です。

- ***Check*** は，チェックボックスの状態を強制的にチェックつきにします。

```
CheckBox->Check();
```

- ***Uncheck*** は，チェックボックスの状態を強制的にチェックなしにします。

```
CheckBox->Uncheck();
```

- ***Toggle*** は，チェックボックスの状態をチェックつきからチェックなしに，またその逆に切り換えます。3つの状態を持つボタンの場合，***Toggle*** はチェックなしのボックスをチェックつきに，チェックつきのボックスを淡色表示に，淡色表示のボックスをチェックなしにそれぞれ変更します。

```
CheckBox->Toggle();
```

- ***SetCheck*** を使うと，チェックボックスの状態を詳細に制御できます。

```
CheckBox->SetCheck(BF_UNCHECKED);    // 選択ボックスをチェックなしにする
CheckBox->SetCheck(BF_CHECKED);      // 選択ボックスをチェックする
CheckBox->SetCheck(BF_GRAYED);       // 3状態選択ボックスを淡色表示にする
```

以上のメンバ関数を1つ呼び出して ***BS_AUTORADIOBUTTON*** スタイルのラジオボタンをチェックすると，同一グループ内でチェック状態にあったラジオボタンのチェックマークが消えます（216ページの「**グループボックス**」を参照）。1つのグループ内のラジオボタンは同時に1つしかチェックできないきまりになっているからです。

チェックボックスメッセージとラジオボタンメッセージへの応答

ユーザがチェックボックスまたはラジオボタンをクリックすると，***BN_CLICKED*** 通知コードのついた通知メッセージが親ウィンドウに送られます。他のコントロールクラスのインスタンスと同様に，このメッセージは通常，親ウィンドウの子ID応答メンバ関数で処理されます。しかし，***TButton*** や ***TCheckBox*** からクラスを派生させ，そのインスタンスに押下時の処理を行なわせることもできます。

グループボックス

Windowsのグループボックス要素はラベルつきの矩形です。これは他のコントロールを見かけ上対応づけるだけで，他のことは何もしません。コントロールを論理グループにまとめて自動ラジオボタンの相互排他関係を決めるのは，グループボックス要素内でのコントロールの位置とは無関係です。論理グループはコントロールの作成順と作成スタイル ***WS_GROUP*** を付与してあるかどうかによって決まります。

ObjectWindowsの ***TGroupBox*** オブジェクトの場合は，単なる「見かけ」だけではなく，コントロールを論理グループにまとめるのに実際に役に立ちます。選択ボックスに対して ***TGroupBox*** を構築時に指定することによって，「グループ」として応答メソッドを定義することができます。すわなち，個別に応答メンバ関数を定義する必要がありません。***TGroupBox*** がグループ内の選択ボックスから受け取ったメッセージを，親への「グループ」通知メッセージに変換するためです。

グループボックスの構築

TGroupBox コンストラクタには，親ウィンドウへのポインタ，コントロールID，位置とサイズのデータという通常のコントロールコンストラクタの引数を使えます。さらに，***LPSTR*** 型の ***Text*** 引数も渡されます。***Text*** は，次のようにグループボックスのラベルになります。

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box",
                            38, 102, 176, 108);
```

グループボックスメッセージへの応答

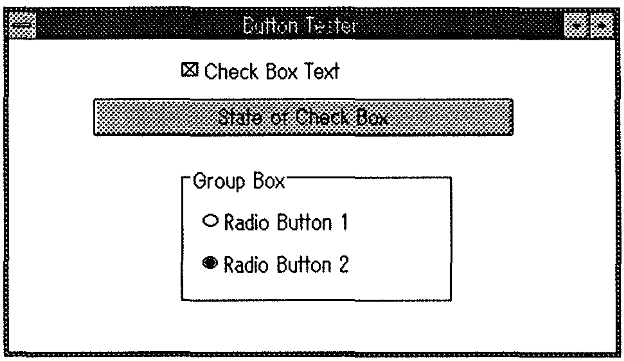
グループに属する選択ボックスのチェック状態が変わると，***TGroupBox*** に通知されます。すると ***TGroupBox*** は，「グループ」通知メッセージをその親に送ります。親はメッセージへの応答として子ID応答メンバ関数を定義し，グループボックスからの通知メッセージを処理させます（グループボックスの識別子は拡張子で指定します）。

プログラム例 GBOXTEST.CPP では，親ウィンドウのメンバ関数を定義して，ラジオボタングループ内のそれぞれの選択変更に応答させます。GBOXTEST.CPP は EXAMPLES ディレクトリに収容されています。

プログラム例：BtnTest

BtnTest プログラムは，プッシュボタン，チェックボックス，ラジオボタン，グループボックスといったコントロールのついたウィンドウを作成します。アプリケーションが起動すると，コントロールがメインウィンドウに表示されます。ユーザがコントロールをクリックすると，アプリケーションは様々な方法で応答します。**図 12.9** を参照してください。

図 12.9
種々のボタンつきのウィンドウ



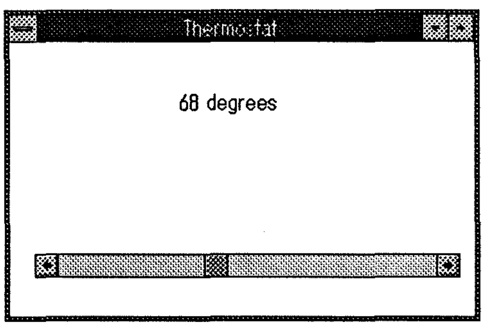
このファイルは BTNTEST.CPP という名前で EXAMPLES サブディレクトリに収めてあります。

スクロールバー

スクロールバーコントロールには，スクロールボックスと呼ばれる移動ボックスがついており，スクロールバーの範囲内を移動することができます。スクロールバー内の各位置は，ユーザが指定した範囲内の特定の位置に対応しています。スクロールバーコントロールは，ユーザが設定した範囲の値を取得するためのものです。たとえば，スクロールバーコントロールで，温度計の設定値や色の値をユーザから受け取ることもできます。

スクロールバーコントロールは，ウィンドウのスクロールバーと外観が似ており，まったく見分けがつかないこともあります。しかし，ウィンドウのビューをスクロールするウィンドウスクロールバーはウィンドウの一部で，子ウィンドウではありません。Windowsでは，作成スタイル ***WS_HSCROLL*** や ***WS_VSCROLL*** が指定されたウィンドウにウィンドウスクロールバーを作成します。ObjectWindowsでは，他のコントロールを作成するのと同じように，スクロールバーコントロールを作成します。親のコンストラクタでスクロールバーオブジェクトを構築するだけで作成できます。

図 12.10
スクロールバーコントロール



スクロールバーオブジェクトの構築

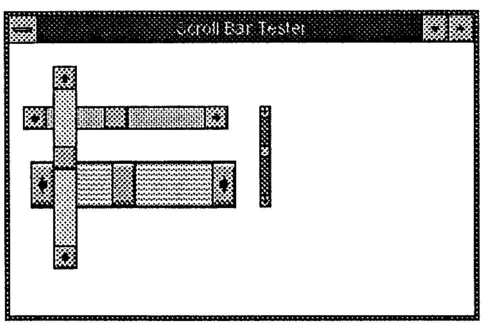
TScrollBar コンストラクタは，通常のコントロールコンストラクタの引数である親ウィンドウを指すポインタ，コントロールID，位置とサイズのデータを使います。さらに，論理引数 *IsHorizontal* も渡されます。この引数で親ウィンドウ内でのスクロールバーの方向（水平か垂直）を指示します。垂直スクロールバーに幅0が指示されると，リストボックスの場合と同様，標準幅のスクロールバーが構築されます。水平スクロールバーに高さ0を指定した場合も同様です。次のコードは，**図 12.10** の標準高さの水平スクロールバーを作成するものです。

```
Thermometer = new TScrollBar(this, ID_THERMOMETER,
                             20, 170, 340, 0, TRUE);
```

TScrollBar コンストラクタは ***TControl*** コンストラクタを呼び出してから，スクロールバーに設定されているデフォルトスタイルに ***SBS_HORZ*** または ***SBS_VERT*** を追加します。スクロールバーのデータメンバ *Attr.Style* を変更すれば，***SBS_TOPALIGN*** などのスタイルも指定することができます。

各種のスクロールバーコントロールを**図 12.11** に示します。

図 12.11
各種のスクロールバーがあるウィンドウ



スクロールバーの範囲は，スクロールバーを作成するときにデフォルトで1から100に設定されます。この設定範囲を変更するには，220ページの節「**スクロールバーの変更**」で説明する ***SetRange*** メンバ関数を使います。

スクロールバーオブジェクトの属性として，行スクロール単位とページスクロール単位もあります。***LineMagnitude*** は，ユーザがスクロールバーのどちらかの端で矢印をクリックしたときにスクロールボックスが移動する量を示します。***PageMagnitude*** は，ユーザがスクロール領域をクリックしたときにスクロールボックスが移動する量を示します。スクロール領域とは，スクロールボックスとスクロールバーの各端との間にあるスペースのことです。***LineMagnitude*** と ***PageMagnitude*** はデフォルトで，それぞれ1と10に設定されます。***TScrollBar*** のデータメンバ ***LineMagnitude*** と ***PageMagnitude*** を直接変更すれば，これらの値を設定し直すことができます。

スクロールバーの照会

スクロールバーの照会用に，***GetRange*** と ***GetPosition*** という2つのメンバ関数が ***TScrollBar*** に定義されています。メンバ関数 ***GetRange*** は，2つの整数参照をとる手続きです。この2つの整数には，スクロールバーの範囲内におけるスクロールボックス位置の最大値と最小値が入ります。

GetPosition は，スクロールボックスの整数位置を返す関数です。プログラムで範囲と位置を取得し，両者を比較するときに使うと便利です。

スクロールバーの変更

ほとんどの場合，スクロールバーはユーザが操作して変更するものですが，プログラムで変更することも可能です。

SetRange は，範囲内の最小値と最大値を示す整数引数をとるメンバ関数です。新しいスクロールバーは，デフォルトで1から100の範囲を持ちます。ただし，スクロールバーで実際に制御する対象に合った範囲に変更することもできます。変更する場合は，スクロールバーを作成してからスクロールバーのメンバ関数 ***SetRange*** を呼び出します。この変更は親のメンバ関数 ***SetupWindow*** で行なえます。なお，***SetRange*** を呼び出す前に親の基本クラスのメンバ関数 ***SetupWindow*** を呼び出して，スクロールバーコントロールを作成しておいてください。

次の例では，温度計プログラムに利用するスクロールバーの範囲を，親のメンバ関数 ***SetupWindow*** で変更しています。

```
void TParentWindowType::SetupWindow()
{
  TWindow::SetupWindow();
  Thermometer->SetRange(32, 120);
}
```

スクロールバーが ***TScrollBar*** からの派生クラスのインスタンスである場合，範囲の変更は子ウィンドウのメンバ関数 ***SetupWindow*** で行なうべきです。

SetPosition は，1つの整数引数をとるメンバ関数です。この引数で，スクロールボックスの移動先となる位置を示します。上で述べた温度計プログラムでは，次のようにして温度を78度に直接設定することができます。

```
Thermometer->SetPosition(78);
```

3番目のメンバ関数 ***DeltaPos*** は，整数引数で指定した量だけスクロールボックス位置を上（左）または下（右）に移動します。正の整数でスクロールボックスを下（右）に移動し，負の整数で上（左）に移動します。たとえば，温度を5度下げるには次のようにします。

```
Thermometer->DeltaPos(-5);
```

スクロールバーイベントへの応答

スクロールバーがスクロールすると，親ウィンドウはスクロールバー通知メッセージを受け取ります。作成した子ウィンドウをスクロールイベントに応答させるときは，通常どおり子ID応答メンバ関数を定義します。ただし，スクロールバー通知メッセージは他のコントロール通知メッセージとは少々異なり，***WM_COMMAND*** ではなく，***WM_HSCROLL*** と ***WM_VSCROLL*** という Windows メッセージに基づいています。そこで唯一注意すべき点は，スクロールバー通知コードが *Msg.LP.Hi* ではなく *Msg.WParam* に格納されるという点です。***SB_LINEUP***，***SB_LINEDOWN***，***SB_PAGEUP***，***SB_PAGEDOWN***，***SB_THUMBPOSITION***，***SB_THUMBTRACK*** は共通です。

ほとんどの場合，すべてのスクロールバーイベントに対し同じ方法で応答します。すなわち，現在の位置を取得し，その位置に応じた動作をします。したがって通常は，次の例のように通知コードを無視してしまいます。

```
void TTestWindow::HandleThermMsg(RTMessage)
{
  int NewPos;
  NewPos = Thermometer->GetPosition();
  // NewPosに基づいて処理
}
```

ユーザが別の位置を選択するまでスクロールボックスのドラッグに応答しないという方法もあります。この場合，***SB_THUMBTRACK*** コードでメッセージを除去してください。

```
void TTestWindow::HandleThermMsg(RTMessage)
{
  int NewPos;
  if (Msg.WParam != SB_THUMBTRACK)
  {
    NewPos = Thermometer->GetPosition();
    // NewPosに基づいて処理
  }
}
```

スクロールバー通知メッセージへの応答をスクロールバーオブジェクト自体にさせたい場合もあります。このような場合は，特定の応答処理をスクロールバーオブジェクトに組み込みます。通知別の応答メンバ関数をクラスに定義すれば，スクロールバーオブジェクトを直接応答させることができます。バーチャルメソッド宣言として，***NF_FIRST*** とスクロールバー通知コードの和を使います。次のように ***TScrollBar*** から新しいクラスを派生させてください。

```
class TSpecializedBar : public TScrollBar
{
public:
  virtual void SBTop(RTMessage Msg) =
    [NF_FIRST + SB_TOP];
};
TSpecializedBar::SBTop(RTMessage Msg)
{
  TScrollBar::SBTop(Msg);
  // スクロールバーが最上部にきたときの専用処理
}
```

メンバ関数 ***SBTop*** が最初に ***TScrollBar*** :: ***SBTop*** を呼び出します。***TScrollBar*** :: ***SBTop*** は，***SB_TOP*** 通知メッセージを受信すると，スクロールボックスを移動させます。***TScrollBar*** は次の表に示すメンバ関数を定義して，各通知メッセージに応答させます。

表 12.5
TScrollBar が定義するメンバ関数

TScrollBar のメンバ関数	通知メッセージ
SBLineUp	*SB_LINEUP*
SBLineDown	*SB_LINEDOWN*
SBPageUp	*SB_PAGEUP*
SBPageDown	*SB_PAGEDOWN*
SBThumbPosition	*SB_THUMBPOSITION*
SBThumbTrack	*SB_THUMBTRACK*
SBTop	*SB_TOP*
SBBottom	*SB_BOTTOM*

以上のメンバ関数のいずれかを定義し直す場合は，定義し直した ***TScrollBar*** のメンバ関数を呼び出して，スクロールボックスを移動してください。

プログラム例：SBarTest

SBarTest は，**図 12.10** に示した温度計アプリケーションを作成するプログラムです。

このファイルは SBARTEST.CPP という名前で EXAMPLES サブディレクトリに収めてあります。

コントロールデータの転送

多数の子ウィンドウコントロールを持つ複雑なダイアログボックスやウィンドウを管理する場合，コントロールの状態を格納したり取得したりするための下位クラスを作成するのが普通です。コントロールの状態として，エディットコントロールのテキスト，スクロールバーの位置，ラジオボタンがチェックされているかどうかがあります。あるいは，ウィンドウやダイアログのコントロールの状態を示す構造体を定義して準備しておくという方法もあります。こうすれば，下位オブジェクトを定義する必要がなくなります。

たとえば，プログラムでモードつきダイアログボックスを表示し，このダイアログボックスをクローズした後でそれぞれのコントロールの状態に関する情報を転送バッファから取り出すことができます。後でユーザがダイアログボックスを再表示したとき，転送バッファに変更がなければ，コントロールはダイアログが最後にクローズされたときの状態になっています。また，転送バッファのデータを初期化すれば，各コントロールを初期状態に設定することもできます。以前の値にコントロールの状態を設定し直す場合などは，どちらかの方向を明示していつでもデータを転送することができます。

転送バッファの定義

転送バッファは，転送に関与する各コントロールごとにフィールドが1つずつある構造体です。コントロールの中には，転送機構に影響されないものもあります。たとえば，状態を持たないプッシュボタンやグループボックスは転送に関与しません。

転送バッファを定義するには，ダイアログやウィンドウ内にある転送に関与する各コントロールに対してそれぞれのデータメンバを定義します。ダイアログ内またはウィンドウ内のすべてのコントロールに対して転送データメンバを定義する必要はなく，値を転送させたいデータメンバだけを定義してください。次に示す転送バッファは，データを転送する各種のコントロールごとに転送データを格納するものです。

```
struct SampleTransferStruct {
  char StaticText[STATICTEXTLEN];      // スタティックテキスト
  char EditText[EDITTEXTLEN];          // エディットテキスト
  TListBoxData ListBoxData;            // リストボックス文字列と選択
  TComboBoxData ComboBoxData;          // コンボボックス文字列と選択文字列
  WORD CheckState;                     // チェックボックス状態
  WORD RadioState;                     // ラジオボタン状態
  TScrollBarData ScrollBarStruct;      // スクロールバー範囲など
}
```

転送バッファに定義するデータメンバの種類はコントロールの種類によって決まります。これは，以下のようにそれぞれのコントロールが異なるデータを転送するためです。

- スタティックコントロールとエディットコントロールは，それぞれのテキストを転送します。データメンバ***TextLen***には，最後のヌルを含む転送テキストの長さが入ります。リストボックスは，***TListBoxData*** 型のオブジェクトを使って，リスト内の文字列，選択された文字列または文字列群，選択された文字列の数を転送します。転送バッファには，次のように，***TListBoxData*** オブジェクトを指すポインタを入れることができます。

```
Struct MyTransferBuffer
{
:
  PTListBoxData pMyListBoxData;
:
};
```

pMyListBoxData は，次のように設定します。

```
MyTransferBuffer.pMyListBoxData = new TListBoxData ();
```

TListBoxData のクラス定義には，次の表に示す3つのデータメンバがあります。

表 12.6
TListBoxData データメンバ

データメンバ	定義
PArrayStrings	リストボックス内の項目を入れた（コンテナクラスライブラリからの）*String* 配列。
PArraySelStrings	リストボックス内の項目を入れた *String* 配列。単一選択リストボックスの *String* は 1 つだけです。
IntSelCount	選択項目数。単一選択リストボックスの *SelCount* は 1 です。

■ コンボボックスは，コンボボックスのリストボックス部分にある文字列と選択された文字列を指すポインタを ***TComboBoxData*** 型のオブジェクトで転送します。リストボックスの場合と同様に，***TComboBoxData*** オブジェクトを指すポインタを転送バッファに含めることができます。ただし，リストボックスには ***TListBoxData*** を使いましたが，コンボボックスには次のように ***TComboBoxData*** を使います。

```
struct MyTransferBuffer
{
  ⋮
  PTComboBoxData pMyComboBoxData;
  ⋮
};
```

pMyComboBoxData は次のように設定してください。

```
MyTransferBuffer.pMyComboBoxData = new TComboBoxData();
```

TComboBoxData のクラス定義には，次の表に示す 2 つのデータメンバがあります。

表 12.7
TComboBoxData のデータメンバ

データメンバ	定義
PArrayStrings	コンボボックスのリストボックス部分にある項目を入れた（コンテナクラスライブラリの）*String* 配列。
PcharSelection	コンボボックスのリストボックスで選択された項目の文字列を指すポインタ。

■ チェックボックスとラジオボタンは，チェック状態を ***WORD*** 値 ***BF_UNCHECKED***，***BF_CHECKED***，***BF_GRAYED*** で転送します。

■ スクロールバーは，***TScrollBarData*** レコードにスクロールバーコントロールの範囲と位置を格納します。***TScrollBarData*** の定義を以下に示します。

```
struct TScrollBarData {
  int LowValue;
  int HighValue;
  int Position;
}
```

コントロールの構築と転送機構の使用

転送機構を使うウィンドウやダイアログでは，転送に関与するコントロールオブジェクトの構築が必要です。構築は，対応する転送バッファのデータメンバを定義した順序で行ないます。ウィンドウやダイアログのオブジェクトで転送機構を使うには，定義した転送バッファを指すようにデータメンバ ***TransferBuffer*** を設定するだけです。

コントロールオブジェクトをコントロール要素に対応させる方法については，第11章『ダイアログオブジェクト』を参照してください。

転送機構では，データの転送先とするコントロールを示すのに ObjectWindows オブジェクトを使います。オブジェクトをダイアログボックス内のコントロールに対応づけるには，親ウィンドウとリソース識別子だけを引数にとるコンストラクタを使います。以下に例を示します。

```
TParentWindow::TParentWindow(PTWindowsObject AParent, LPSTR ATitle) :
                            TWindow(AParent, ATitle)
{
  TheDialog = new TDialog(this, ID_DIALOG);
  new TStatic(TheDialog, ID_STATIC, 20);
  new TEdit(TheDialog, ID_EDIT);
  new TListBox(TheDialog, ID_LISTBOX);
  new TComboBox(TheDialog, ID_COMBOBOX, 20);
  new TCheckBox(TheDialog, ID_CHECKBOX, 0);
  new TRadioButton(TheDialog, ID_RADIOBUTTON, 0);
  new TScrollBar(TheDialog, ID_SCROLLBAR);
  TheDialog->TransferBuffer = &TransferStruct; }
```

デフォルトでは，TStatic のインスタンスはダイアログの内容も転送しません。

ウィンドウとダイアログではデータの転送方法にもう1つ違いがあります。ウィンドウのコントロールでは，デフォルトで転送機構が使えません。一方，ダイアログのコントロールでは，***TStatic*** のインスタンス以外は，デフォルトで転送機構が使えます。ウィンドウのコントロールにも転送機構を使えるようにするには，次のように，メンバ関数 ***EnableTransfer*** を呼び出します。

```
  ⋮
Edit = new TEdit(this, ID_EDIT, "", 10, 10, 100, 30, 40, FALSE);
Edit->EnableTransfer();
  ⋮
```

コントロールを転送機構から明示的に排除するには，メンバ関数 ***DisableTransfer*** を呼び出します。

データの転送

ウィンドウやダイアログを作成したり，またはダイアログを実行すると，転送バッファから転送に関与するコントロール群にデータが自動的に転送されます。

モードつきダイアログの場合，コントロールIDが ***IDOK*** になっているコマンドメッセージをダイアログが受信すると，データがコントロールから転送バッファに自動的に転送されます。***CloseWindow*** から ***CanClose*** への呼び出しで真が返ると，データがモードつきダイアログのコントロールから自動的に転送バッファに転送されます。ダイアログが再び実行されると，バッファのデータは再びコントロールに転送されます。この転送方式でダイアログとバッファは同期しています。

データの転送はどちらの方向でも明示的にいつでも行なえます。たとえば，ウィンドウまたはモードなしダイアログのコントロールからデータを転送したい場合があります。または，ユーザがResetボタンを押したときに，転送バッファにあるデータを使ってコントロールの状態を設定し直したい場合もあります。このような場合にはメンバ関数 ***TransferData*** を使います。定数 ***TF_SETDATA*** を渡すと，バッファからコントロールに転送されます。また，定数 ***TF_GETDATA*** を渡すと反対の方向に転送されます。たとえば，次のようにしてウィンドウオブジェクトのメンバ関数 ***CloseWindow*** で ***TransferData*** を呼び出すこともできます。

```
void TSampleWindow::CloseWindow()
{
  TransferData(TF_GETDATA);
  TWindow::CloseWindow();
}
```

コントロール転送のカスタマイズ

特定のコントロールのデータ転送方法を変更したり，新しく定義したコントロールを転送機構に取り入れたい場合があります。どちらの場合でも，コントロールオブジェクトに対してメンバ関数 ***Transfer*** を記述するだけのことです。このメンバ関数は ***TF_GETDATA*** が渡されると，渡されたアドレス位置にコントロールからデータをコピーします。***TF_SETDATA*** が渡されると，渡されたアドレスのデータをコントロールにコピーします。***TF_SIZEDATA*** が渡されると，転送データのサイズを返し，後はなにもしません。転送データのサイズは，***TStatic*** のデータメンバ ***TextLen*** に入ります。***TStatic***::***Transfer*** の例を以下に示します。

```
WORD TStatic::Transfer(Pvoid DataPtr, WORD TransferFlag)
{
  if (TransferFlag == TF_GETDATA)
    GetText((LPSTR)DataPtr, TextLen);
  else
    if (TransferFlag == TF_SETDATA)
      SetText((LPSTR)DataPtr);
  return TextLen;
}
```

メンバ関数 ***Transfer*** は，必ず転送したバイト数を返します。

転送例

TranTest プログラムのメインウィンドウは，フィールドつきのモードつきダイアログを作成します。ユーザはこのフィールドに名前とアドレス情報を入力することができます。このプログラムはこれらの情報を転送バッファに格納し，ダイアログが再び実行されたときにダイアログのコントロールに表示します。なお，新しいダイアログクラスはダイアログのデータを設定したり取り出したりするように定義していません。

TRANTEST.CPP は，EXAMPLES サブディレクトリに収めてあります。また，LBXTTEST.CPP と CBXTTEST.CPP は，リストボックスとコンボボックスのデータ転送を行なうプログラム例です。この 2 つのプログラムは，この章の前半で紹介したプログラム LBOXTEST.CPP と CBOXTEST.CPP に変更を加えたものです。

第 13 章

カスタムコントロールオブジェクト

ObjectWindows コントロールクラスのデフォルトの特性，外観，動作についてはすでに紹介しました。デフォルトを変更する方法も一部取り上げました。この章では，ObjectWindows の定義済みのコントロールの特性を変える方法を詳しく説明します。独自のコントロールを設計する方法も紹介します。

定義済みコントロールの変更

定義済みの ObjectWindows コントロールの特性を変更する基本的な方法は，次の 2 つです。

- 作成スタイルの変更
- メッセージ応答メンバ関数の定義

作成スタイルの変更

Attr.Style が保持するスタイル作成属性を変更すれば，定義済みのコントロールのデフォルト動作を変更することができます。たとえば，ラジオボタンの選択を相互排他的にしたくなければ，***BS_AUTORADIOBUTTON*** を作成スタイルから除去し，かわりに ***BS_RADIOBUTTON*** を追加します。なお，こうするとボタンのチェックが自動でなくなるのでプログラム側でチェックしなければなりません。また，リストボックスから複数の項目を選択できるようにするには，リストボックスの作成スタイルに ***LBS_MULTIPLESEL*** を追加します。

作成スタイルを変更すれば，定義済みのコントロールの外観をある程度まで変えることができます。たとえば，次のように指定すると，チェックボックスコントロールのテキストがボックスの左側に表示されます。

```
MyBox = new TCheckBox(TheParent, ID_CHECKBOX, "Text",
                      100, 100, 80, 20, NULL);
MyBox->Attr.Style |= BS_LEFTTEXT;
```

コントロールに枠をつけたければ，その作成スタイルに ***WS_BORDER*** を追加します。

定義済みのコントロールの中には，**オーナー描画**作成スタイルを指定してコントロールの外観を大幅に変更できるものもあります。**描画可能**コントロールは次の節で取り上げます。

定義済みのコントロール特性を作成スタイルの変更によって変える方法には限度があります。コントロールの種類ごとに使えるスタイルオプションが決まっているためです。各種コントロールの作成スタイルオプションの一覧は **ObjectWindows for C++リファレンスガイド**にあります。

定義済みのコントロールを描画可能にする

定義済みコントロールの中には，オーナー描画スタイルを指定してコントロールの外観をある程度まで定義できるものがあります。たとえば，次のように ***BS_OWNERDRAW*** という作成スタイルを指定すると，ボタンの描画が可能になります。

```
Attr->Style |= BS_OWNERDRAW;
```

描画方法の定義も必要な場合は，派生クラスでメンバ関数 ***ODADrawEntire*** を定義します。***ODAFocus*** と ***ODASelect*** を定義し直せば，描画可能コントロールがフォーカスを受けたり選択された場合に外観を変えることができます。

メンバ関数 ***ODADrawEntire***，***ODAFocus***，***ODASelect*** には，コントロールの描画情報が入った ***DRAWITEMSTRUCT*** が渡されます。データメンバ *hDC* は使用するディスプレイコンテキストへのハンドルが入ります。このハンドルは，コントロールを描画するときに呼び出される Windows GDI 関数に必ず渡してください。また，データメンバ *rcItem* には，コントロールのクライアント領域があります。***DRAWITEMSTRUCT*** の詳細については **ObjectWindows for C++リファレンスガイド**の第 6 章『**Windows 型**』を参照してください。

描画可能コントロールは，***WMDrawItem*** を定義し直した親ウィンドウが描画することもできます。この場合，***DRAWITEMSTRUCT*** を指すポインタが，指定した ***RTMessage*** の ***LParam*** として渡されます。すなわち，描画するコントロールの識別子が構造体のメンバ ***CtlID*** で渡されます。

EXAMPLES ディレクトリにあるアプリケーション DCTLTEST.CPP は，メニュー項目「Test」が選択されると，描画可能なボタンコントロールつきのダイアログを表示します。このダイアログを表示して，フォーカスと選択を変えたときにどのように描画可能ボタンが応じるかを見てください。フォーカスは **TAB** と**矢印**キーで移動させます。ボタンを選択するときはクリックします。描画可能ボタンに対して，アプリケーションでビットマップを使うこともできます。

BS_OWNERDRAW という作成スタイルを指定すると，プッシュボタン，チェックボックス，ラジオボタン，グループボックスが描画できるようになります。リストボックスとコンボボックスを描画できるようにするには，作成スタイル ***LBS_OWNERDRAW*** と ***CBS_OWNERDRAW*** をそれぞれ指定します。Windows ではスクロールバー，エディットコントロール，スタティックコントロールは描画できません。

定義済みのメッセージ応答の変更

定義済みのコントロールが受信する Windows メッセージに対し，独自の応答を定義できます。カスタマイズするには，定義済みコントロールクラスからクラスを派生させ，そこでメッセージ応答用のメンバ関数を定義します。このやり方で，定義済みのコントロールの外観や動作を変更することもできます。

定義済みコントロールに対する追加処理

定義済みコントロールが受信するメッセージの処理に付加処理を追加できます。付加処理の追加は ***DefWndProc*** を呼び出して定義済み応答を起動する前でも後でもかまいません。

たとえば，ボタンがクリックされたときに親に知らせるだけでなくビープ音も鳴らすには，***WMLButtonDown*** を次のように定義し直します。

```
void TBeepButton::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

ボタンに対する ***WM_LBUTTONDOWN*** のデフォルト処理は通知メッセージ ***BN_CLICKED*** の送出であるので，別の実現方法として ***BNClicked*** を定義し直すこともできます。

別の例として，メンバ関数 ***WMSetFocus*** と ***WMKillFocus*** を定義して，フォーカスが当てられたエディットコントロールの枠を変更することもできます。この場合も，デフォルト処理で通知メッセージ ***ENSetFocus*** と ***ENKillFocus*** が送出されるので，メンバ関数 ***ENSetFocus*** と ***ENKillFocus*** を定義することもできます。

定義済みコントロールの応答のオーバーライド

定義済みコントロールのデフォルト処理を，ある程度までオーバーライドすることができます。ただし，このオーバーライドは予想以上に手がかかります。コントロールのデフォルト処理が複雑な場合があるからです。オーバーライドした後は必ず全体をテストしてください。定義済みの特定の動作をシミュレートしなければならない場合もあるはずです。

たとえば，読み込み専用のエディットコントロールを実現するには，***TEdit*** からの派生クラスで「変化なし」を示す ***WMChar*** 応答をさせます。このようなエディットコントロールではテキストを表示し，これをクリップボードにコピーすることはできても変更することはできません。

定義済みの動作をオーバーライドする別の例として，ボタンの ***WMLButtonDown*** の定義済み処理をオーバーライドして，クリックされても外観を変えない「無反応」ボタンを実現する場合があります。この場合，***BN_CLICKED*** コードのついた通知メッセージを明示的にボタンの親に送らなければなりません。これは ***WMLButtonDown*** の定義済み処理では通常，自動的に送られるものだからです。

```
void TFlatButton::WMLButtonDown(RTMessage Msg)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, BN_CLICKED));
}
```

さまざまな応答をカスタマイズできるようになったら，そろそろ独自のコントロールを最初から定義してみましょう。

カスタムコントロールの使用

コントロールに要求する動作が定義済みのコントロールの動作と異なる場合は，カスタムコントロールを使うことになります。カスタムコントロールではコントロールのすべての動作だけではなく，その外観も定義します。

カスタムコントロールの設計

カスタムコントロールを設計する基本的なステップは次のとおりです。

- 取得するユーザ入力の種類を特定する。
- コントロールの外観を定義する。
- 入力に対する応答を定める。

単純なコントロールであれば，どれもマウスの左ボタンのクリックに応答し，通知メッセージを親に送ります。アプリケーションで使うコントロールは入力の取得や応答をどのようにでも設計できますが，使いやすくて一貫したインターフェースにすることが目的です。したがって，ある程度は約束事を守ることが必要です。

カスタムコントロールの定義

カスタムコントロールを実現する基本的なステップは次のとおりです。

- ***TControl*** からの派生クラスを宣言する。
- ***Paint*** を定義し直して，コントロールを描画させる。
- ユーザ入力を受け取って応答するメッセージ応答メンバ関数を定義する。

カスタムコントロールは，EXAMPLESディレクトリにあるアプリケーション例CCTLTEST.CPPで使われています。次のカラーコントロールは，ユーザが選択できるカラーオプションを表示します。*TColorControl* オブジェクトは左ボタンのクリックに応答して，親に通知メッセージを送ります。

```
void TColorControl::WMLButtonDown(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, CN_CLICKED));
}

void TColorControl::WMLButtonDblClk(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, CN_DBLCLICKED));
}
```

カラーコントロールの親に渡される定数 ***CN_CLICKED*** と ***CN_DBLCLICKED*** は，発生したイベントの種類（クリックかダブルクリック）を記述した通知コードです。カスタムコントロールはカスタム通知コードつきの通知メッセージを送りますが，親が認識できるコードであればどのようなコードでも送れます。このような通知はコントロールからの標準の通知はでないので，次のように定義する必要があります。

```
const unsigned int CN_CLICKED = 0;
const unsigned int CN_DBLCLICKED = 1;
```

アプリケーション *CCTLTEST* の *TColorControl* は，該当する色のブラシでクライアント領域を描画するように ***Paint*** を定義し直しています。作成したカスタムコントロールも ***Paint*** を定義し直す必要があるでしょう。見えないコントロールでは役に立たないからです。また，フォーカスを受け取ったり失ったときにコントロールの外観を変えたければ，メッセージ応答メンバ関数 ***WMSetFocus*** と ***WMKillFocus*** も必要です。

デフォルトの属性，およびGetWindowClassとGetClassNameの詳細については，第10章『ウィンドウオブジェクト』の「ウィンドウクラスの登録」を参照してください。

TContorl は ***TWindow*** からメンバ関数 ***GetWindowClass*** を継承しています。このメンバ関数はカスタムコントロールに合った登録属性をデフォルトとして指定します。これらのデフォルトを変更してカスタムコントロールに使う場合は，***GetWindowClass*** と ***GetClassName*** を定義し直してください。*TColorControl* の例では，次のように登録クラススタイルを変更してコントロールにダブルクリックメッセージを受け取らせています。

```
void TColorControl::GetWindowClass(WNDCLASS& AWndClass)
{
  TControl::GetWindowClass(AWndClass);
  AWndClass.style |= CS_DBLCLKS;
}

LPSTR TColorControl::GetClassName()
{
  return "ColorControl";
}
```

ダイアログのカスタムコントロールのWindowsクラス名は，ダイアログのリソース定義で指定する必要があります。コントロールのWindowsクラス名とは，メンバ関数 ***GetClassName*** が返す名前です。***GetClassName*** を定義し直していないと，“OWLWindow”が返されます。

第14章

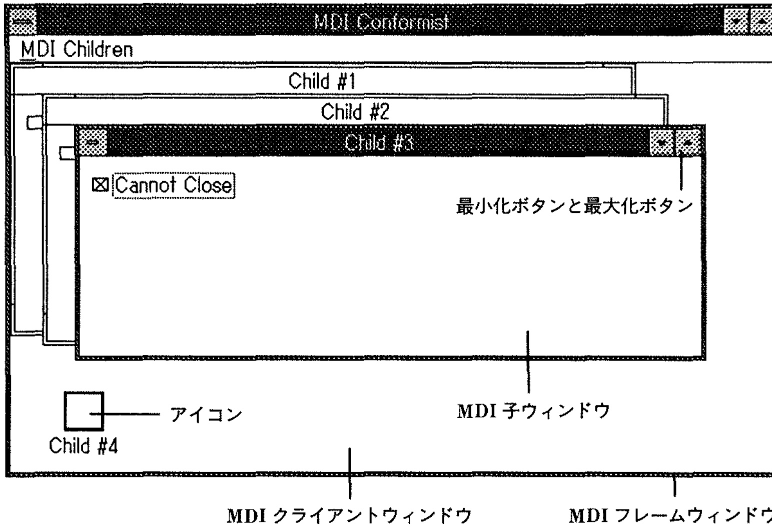
MDIオブジェクト

マルチドキュメントインターフェース（MDI）は，ユーザがWindowsアプリケーションで複数のドキュメントを同時にオープンして処理するためのインターフェース基準です。この意味で，ドキュメントとは，テキストファイルの編集とか表計算ファイルの処理といったファイル固有のタスクということになります。MDIに準拠したアプリケーションのユーザは，1つのアプリケーション内に複数のファイルをオープンすることなどができます。MDIアプリケーションの例としては，Windowsプログラムマネージャ，Windowsファイルマネージャなどを挙げることができます。MDIは，IBMのCommon User Access仕様の一部にもなっています。

MDIアプリケーションの要素

各MDIアプリケーションには共通の要素がいくつかあります。最も一般的なのが，**MDIフレームウィンドウ**というメインウィンドウです。このフレームウィンドウのクライアント領域には，**MDIクライアントウィンドウ**という見えないウィンドウがあり，**MDI子ウィンドウ**という動的に作成される子ウィンドウを背後で管理しています。

図 14.1
MDI アプリケーションの要素例



フレームウィンドウにはメニューがあります。ユーザはこのメニューの項目 Tile, Cascade, Arrange, Close All などを選択して MDI 子ウィンドウを操作することができます。Window というタイトルがつけられることが多いこのメニューは**子ウィンドウメニュー**と呼ばれます。オープンした MDI 子ウィンドウは，順次このメニューの最後に自動的に追加されていきます。現在選択されているウィンドウはメニュー上でチェックマークがついています。

MDI 子ウィンドウには，オーバーラップウィンドウに似た特徴があります。すなわち，フレームウィンドウの大きさまで最大化することができ，アイコンにまで最小化することができます。MDI 子ウィンドウは，フレームウィンドウの境界枠の外に表示されることはありません。また，MDI 子ウィンドウにはメニューがなく，すべての機能はフレームウィンドウのメニューで制御します。各 MDI 子ウィンドウのタイトルは，多くの場合そのウィンドウに対応してオープンしているファイルの名前です。ただし，MDI 子ウィンドウの動作はプログラムで自由に変更できます。このように，MDI アプリケーションは，多数のアプリケーションがウィンドウやアイコンで表わされた，小規模な Windows セッションと考えることができます。

オブジェクトとしての MDI ウィンドウ

ObjectWindows では ***TMDIFrame*** と ***TMDIClient*** というクラスが定義されており，それぞれのインスタンスで MDI フレームウィンドウと MDI クライアントウィンドウが表わされます。MDI 子ウィンドウは，***TWindow*** の派生クラスのインスタンスです。

ObjectWindows の MDI アプリケーションでは，フレームウィンドウが固有の MDI クライアントウィンドウを所有しており，その参照をデータメンバ ***ClientWnd*** に格納しています。フレームウィンドウはさらに，MDI 子ウィンドウへの参照も子リストに保持しています。

TMDIFrame のメンバ関数の主な目的は，MDI 子ウィンドウと MDI クライアントウィンドウの構築と管理，およびメニュー選択の処理です。***TMDIClient*** の主な役目は，背後から MDI 子ウィンドウを管理することです。MDI アプリケーションを設計する場合，通常フレームと子ウィンドウに対して ***TMDIFrame*** と ***TWindow*** から新しいクラスを派生させることになります。MDI クライアントウィンドウには通常 ***TMDIClient*** のインスタンスを使います。

MDI ウィンドウの構築

MDI アプリケーションのウィンドウを構築する場合には，特に注意すべき点がいくつかあります。ウィンドウの構築の詳細については，第 10 章『**ウィンドウオブジェクト**』を参照してください。

MDI フレームウィンドウの構築

MDI アプリケーションのフレームウィンドウはメインウィンドウでもあるため，そのアプリケーションオブジェクトのメンバ関数 ***InitMainWindow*** の内部から構築されます。ただし，***TWindow*** と異なり，***TMDIFrame*** のコンストラクタは，親ウィンドウを引数として使わずにメニューのリソース ID を引数にします。これは，フレームウィンドウがメインウィンドウであり，親がないからです。MDI フレームウィンドウにはメニューが必要なので，***TMDIFrame*** のコンストラクタはメニュー引数を必要とし，*Attr.Menu* を設定します。

フレームウィンドウのメニューには，必ず MDI スタイルの子ウィンドウメニューを入れます。このメニューには，MDI 子ウィンドウのメニュー項目が追加されます。***TMDIFrame*** のデータメンバ ***ChildMenuPos*** には，子ウィンドウのサブメニューの位置が入ります。***TMDIFrame*** のコンストラクタは ***ChildMenuPos*** の初期値をゼロに設定して，左端の最上位レベルメニュー項目を示します。ただし，MDI ウィンドウのコンストラクタで ***ChildMenuPos*** を設定し直すこともできます。

```
TMyMDIWindow::TMyMDIWindow(LPSTR ATitle, HMENU AMenu) :
                              TMDIFrame(ATitle, AMenu)
{
  ChildMenuPos = 1;
}
```

さらに ***TMDIFrame*** :: ***SetupWindow*** は ***InitClientWindow*** を呼び出して ***TMDIClient*** オブジェクトを構築し，自分の MDI クライアントウィンドウとして使います。***TMDIFrame*** :: ***SetupWindow*** が MDI クライアントウィンドウを作成します。

MDI 子ウィンドウの構築

TMDIFrame は，***CMCreateChild*** という自動応答メンバ関数を定義しています。このメンバ関数は，***CM_CREATECHILD*** ID のついたメニュー項目が選択されると呼び出されます。通常，このメニュー項目は New または Create になります。***TMDIFrame*** によって定義されているため，***CMCreateChild*** は ***TMDIFrame*** :: ***CreateChild*** を呼び出して，MDI 子ウィンドウオブジェクト

を構築します。次に ***CreateChild*** が ***InitChild*** とアプリケーションオブジェクトのメンバ関数 ***MakeWindow*** を呼び出して，ウィンドウを作成します。

TMDIFrame :: ***InitChild*** は ***TWindow*** 型のタイトルなしのウィンドウを作成するだけです。したがって，MDI 子ウィンドウのインスタンスが構築されるように，MDI ウィンドウクラスの ***InitChild*** を定義し直す必要があります。

```
PTWindowsObject TMyMDIWindow::InitChild()
{
  return(new TMyMDIChild(this, "New Child Window"));
}
```

こうすれば，***CreateChild*** が呼び出されたときに，MDI 子ウィンドウクラスのインスタンスが構築され作成されます。また，フレームウィンドウが最初に表示されたときに，MDI 子ウィンドウを **1 つ**表示させることもできます。他の子ウィンドウと異なり，MDI 子ウィンドウの構築と作成はコンストラクタからではなく，MDI フレームウィンドウのメンバ関数 ***SetupWindow*** で行なわなければなりません。さらに，次のように子ウィンドウを明示的に作成しなければなりません。

```
void TMyMDIWindow::SetupWindow()
{
  TMDIFrame::SetupWindow();
  CreateChild();
}
```

この最初の MDI 子ウィンドウを最大化する場合は，直接，または ***InitChild*** を呼び出して MDI 子ウィンドウオブジェクトを構築してから，このオブジェクトのスタイルに ***WS_MAXIMIZED*** を追加します。次に，アプリケーションオブジェクトのメンバ関数 ***MakeWindow*** を呼び出して MDI 子ウィンドウを作成します。

```
void TMyMDIWindow::SetupWindow()
{
  PMyMDIChild NewChild;

  TMDIFrame::SetupWindow();
  NewChild = new TMyMDIChild(this, "New Child Window");
  NewChild->Attr.Style |= WS_MAXIMIZE;
  GetApplication()->MakeWindow(NewChild);
}
```

アプリケーションによっては，複数のメニュー項目に応答するMDI子ウィンドウが必要になる場合があります。たとえば，ファイルエディタのメニュー項目**New**と**Open**の両方が，ファイル名をタイトルに持つ新しい子ウィンドウを表示させる場合などです。この場合，子ウィンドウを構築し作成するように，それぞれのメニュー項目のコマンド応答メンバ関数を定義します。

MDIアプリケーションでのメッセージ処理

通常の，つまりMDIでない親ウィンドウと子ウィンドウの場合，Windowsのコマンドメッセージは最初に子ウィンドウに送られます。一部のメッセージはここで受け取られ処理されます。その後，メッセージは親ウィンドウに送られます。このように，現在アクティブなMDI子ウィンドウでの処理をフレームウィンドウのメニューで制御することができるのです。また，フレームウィンドウでメニューの選択に応答する場合もあります。

MDI子ウィンドウの管理

ObjectWindowsのMDIウィンドウクラスには，MDIアプリケーションのMDI子ウィンドウを操作するためのメンバ関数が提供されています。基本的な処理の多くは***TMDIClient***によって実行されますが，機能やデータはすべてメンバ関数***TMDIFrame***で操作します。

子ウィンドウの起動

MDIアプリケーションのユーザは，オープン中またはアイコン状態のMDI子ウィンドウを自由にアクティブにすることができます。ただし，ユーザが別の子ウィンドウをアクティブにすることによって現在アクティブな子ウィンドウを非アクティブにしたときの対策が必要な場合があります。たとえば，フレームウィンドウのメニューで，淡色表示やチェックマークによってアクティブな子ウィンドウの現在の状態を示す場合です。ユーザが子ウィンドウをアクティブにしたり

非アクティブにしたりするたびに，このウィンドウに Windows メッセージ ***WM_MDIACTIVATE*** が送られます。子ウィンドウオブジェクトに対して ***TMDIClient*** :: ***WMMDIActivate*** を定義し直せば（通常は ***DefWndProc*** を呼び出すのみ），子ウィンドウはこのメッセージに応じることができます。

子ウィンドウメニュー

TMDIFrame は，Tile, Cascade, Arrange Icons, Close All といった標準の MDI メニュー項目の選択に自動的に応答する Windows メッセージ応答メンバ関数を定義しています。このメンバ関数は，定義済みのメニュー ID 定数のついたコマンドメッセージにしか応答しません。したがって，子ウィンドウのメニューリソースを構築するときには必ず次の ID を使ってください。

表 14.1
標準の MDI 処理，コマンド，メンバ関数

処理	メニュー ID 定数	TMDIFrame メンバ関数
Tile	*CM_TILECHILDREN*	*CMTileChildren*
Cascade	*CM_CASCADECHILDREN*	*CMCascadeChildren*
Arrange Icons	*CM_ARRANGEICONS*	*CMArrangeIcons*
Close All	*CM_CLOSECHILDREN*	*CMCloseChildren*

CMTileChildren のような ***TMDIFrame*** の応答メンバ関数は，***TileChildren*** など，***TMDIFrame*** の他のメンバ関数を呼び出します。***TMDIFrame*** のメンバ関数は，さらに ***TMDIClient*** :: ***TileChildren*** など，***TMDIClient*** の同じ名前のメンバ関数を呼び出します。通常，この動作を定義し直すことはありませんが，必要であれば ***TMDIFrame*** :: ***TileChildren*** または ***TMDIFrame*** の他のメンバ関数（自動応答メンバ関数を除く）を定義し直すことができます。

MDI アプリケーションの例

MDITest は，**図 14.1** に示す MDI 準拠のアプリケーションを作成するプログラムです。

このファイルは MDITEST.CPP という名前で EXAMPLES サブディレクトリに収めてあります。

第15章

ストリームオブジェクト

この章では，ObjectWindowsのストリームマネージャがどのようにしてObjectWindowsオブジェクトを**長持ち**させるかについて説明します。ObjectWindowsアプリケーションで作成したウィンドウ，ダイアログボックス，コレクションなどの各種オブジェクトの寿命ははかないものです。アプリケーションの展開とともに，構築され，表示され，呼び出され，破棄されてしまいます。オブジェクトはそれぞれの担当範囲が始まると現われ，終わると隠れ，最後にプログラムが終了すると完全に消え去ります。ストリームマネージャを使えば，オブジェクトをメモリかファイルストリームに保存して，通常の寿命より長持ちさせることができます。

オブジェクトを長持ちさせられれば，多くの用途に使えます。たとえば，共有メモリに保管すれば，プロセス間通信ができます。モデム経由で他のシステムに転送できるからです。ファイルストリームを使えば，ディスクに永久に保存することができます。ディスクに保存したオブジェクトは，同一のアプリケーションでも，同一アプリケーションの別のインスタンスでも，別のアプリケーションでも読み戻して復元することができます。ObjectWindowsオブジェクトのすべてのオブジェクトで効率的で安全なストリーム機能を利用できます。***TButton***，***TDialog***，***TWindow***など，すべてのObjectWindowsクラスはストリームクラスとして設計されているので，通常のファイルI/Oのように簡単にストリームで送ることができます。

独自のストリームクラスを構築するのも簡単です。クラスをストリーム形式にしても，ほとんどオーバーヘッドがかかりません。ストリームクラスには，3つの仮想関数 ***read***，***write***，***streamableName*** の他に，ビルダとビルドコンストラクタが必要です。関数 ***read***，***write***，***streamableName*** は仮想関数として ***TStreamable*** で宣言します。ストリームクラスはすべて，直接または間接的に ***TStreamable*** から派生させてください。***TWindowsObject*** には ***TStreamable*** がすでに基本クラスとして備わっているので，***TWindowsObject*** から派生するクラスはすべてストリーム形式になります。

ストリームオブジェクトは，***pstream*** とその派生クラス群だけで作成できます。これらの派生クラスは，寿命の長いストリームI/Oに特に提供されており，C++の標準 **iostream** クラスによく似ています。したがって，この **iostream** クラスに詳しい方であれば，簡単に習得できるはずです。ストリームを初めて扱かう方を対象に，次の節でストリームを簡単に紹介し，基本的な用語を解説します。

iostream ライブラリ

Cにも C++にも入出力操作を扱うキーワードや定義済みの演算子はなく，標準ライブラリに組み込まれた関数を使っています。Cの場合は **stdio**，C++の場合は **iostream** です。C++の **iostream** ライブラリのほうが，柔軟性，拡張性，安定性に優り，標準データ型とユーザ定義のデータ型の両方にオーバーロード演算子と型保証された入出力を提供します。**printf** などの **stdio** 関数がC++で利用できますが，C++のプログラマの多くはストリームによる方法を利用したがります。ストリームとはいったい何でしょう。

ストリームとは，様々なアクセス属性を持つ項目の無限大（理論上ですが）のシーケンスです。ストリームには，**長さ**（要素数），**現在位置**（ある一時点における唯一のアクセス点），**アクセスモード**（入力用の読み込み専用，出力用の書き込み専用，入出力用の読み書き両用）があります。読み込み（すなわち抽出）は現在位置で行ない，そこから通常は次の項目に進みます。一方，書き込み（すなわち挿入）は，項目をストリームの最後に加えます。

従来のディスクファイルも，ストリームという概念を実現したものです。しかし，この概念は拡大され，メモリのストリーム，キーボードからの文字ストリーム（標準入力 **cin** など），画面への文字ストリーム（標準出力 **cout** と **cerr** など），

シリアルポートやその他のデバイスとの両方向ストリームなども含むようになっています。この拡大概念は主に，「すべてがファイルだ」とするUNIXから生じたものです。事実，OOP技術を利用すると，データの**ソース（生産側）**にも**抽出**メンバ関数を提供でき，入力ストリームとして扱うことができます。同様に，データの**受け手（消費側）**のほとんどに**挿入**メンバ関数を与えることができ，出力ストリームとして扱うことができます。ディスクファイルに対応したストリームは，入力も出力も提供できるのが普通です。C++の標準**iostream**ライブラリについては**Borland C++ユーザーズガイド**を参照してください。このライブラリは，ストリームクラスの広い階層構造を実現して，そのストリームクラスの変形を多数扱うことができます。ストリームクラスは，バッファリングつきかなしか，書式化されているかいないか，メモリ内かファイル上か，などを様々に組み合わせた構成をとり，さらにそれぞれを入力，出力，入出力バージョンで実現できます。C++のObjectWindowsはこの構成上に構築され，ObjectWindowsで使われる一層複雑なオブジェクトに対しストリームI/Oを提供します。

オーバーロード演算子<<と>>

C++のストリームI/Oに人気がある理由として，出力（挿入）用の<<と入力（抽出）用の>>という，連結できるオーバーロード演算子が便利な点が挙げられます。以下に示すように，複雑な構文の**printf**やその他の**stdio**ライブラリ関数が単純ですっきりした文に置き換えられます。

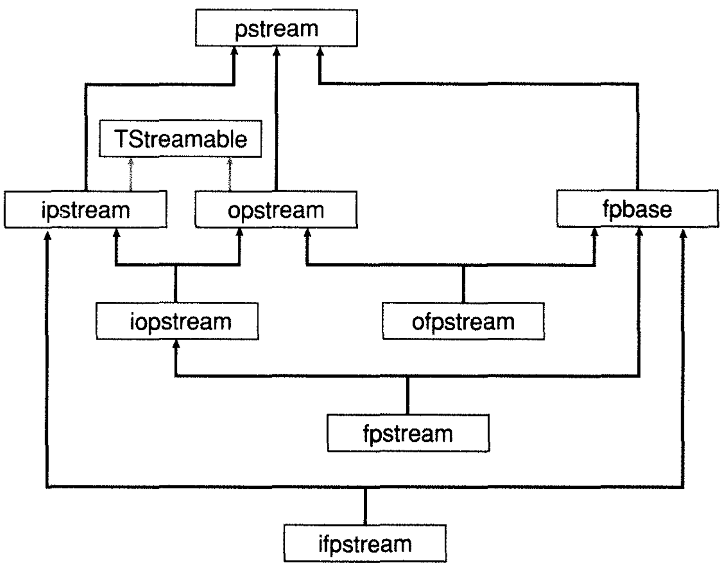
```
cout << "Hello, World" << endl;
```

指定したデータ型とストリームクラスに対して，<<と>>が適切にオーバーロードされれば，その型のオブジェクトを該当するストリームオブジェクトに書き込んだり，そこから読み込んだりできます。このオーバーロード機能は，**char, short, int, long, char*, float, double, void***などC++の標準型に組み込まれています。オーバーロード機能は，ユーザ定義のクラスなしデータ型にも簡単に利用できます。クラスつきオブジェクトにオーバーロード機能を使うのは多少複雑ですが，ObjectWindowsではすでに実現されています。したがって，次のような文でオブジェクトの読み込み書き込みができます。

```
os << AWindow << ADialog;
is >> AButton;
```

この場合，内部の事情を気遣う必要はありません。独自のストリームクラスを作成して**登録**する場合だけいくつか単純な予備作業が伴いますが，これに関してはもう少し後で述べます。上に示したコード例で，***os*** は ***opstream***（または派生）オブジェクトで，***is*** は ***ipstream***（または派生）オブジェクトです。両クラスはいずれも ***pstream*** から派生し，寿命の長い(persistent)オブジェクトに基本のストリームクラスを付与します（両クラス名に p がつくのは "p"ersistent に由来します）。両クラスとも，C++の標準クラス階層にある **ios** から派生した **istream** と **ostream** に似ています。***ipstream*** クラスと ***opstream*** クラスを合わせた ***iopstream*** というクラスもあります。また，***ifpstream***，***ofpstream***，***fpstream*** というファイルストリームの変形もあります。これらは，C++の標準クラス **ifstream**，**ofstream**，**fstream** に対応しており，同じようなフィールドとメンバ関数があります。***pstream*** クラスの階層構造を以下に示します。

図 15.1
ObjectWindows で使用するストリームクラスの階層



たとえば，***TWindow*** のオブジェクトとオブジェクトポインタ ***opstream*** に書き込むためのオーバーロード演算子<<は，次のように WINDOW.H で定義します。

```
inline Ropstream operator << (Ropstream os, RTWindow cl);
   { return os << (RTStreamable) cl; }
inline Ropstream operator << (Ropstream os, PTWindow cl);
   { return os << (PTStreamable) cl; }
```

最初の例では，***TWindow*** オブジェクト *cl* が **opstream** オブジェクト *os* に書き込まれます。同じ出力ストリームが演算子から返されるので，C++の標準クラス **ostream** で使われる<<演算を連結することができます。2番目の変形例では，***TWindow*** オブジェクトを指すポインタに対して同じことを行ないます。なお，対象オブジェクトを型キャストして，基本クラスで定義された<<演算子の呼出しを起動するだけで演算子は実現されます。この場合，***TWindow*** は基本クラス ***TStreamable*** を持ちます。

オーバーロード演算子<<と>>は，同じように ObjectWindows 内のすべての標準ストリームクラスに提供されています。これらは通常インラインで実現され，上の形式に従います。独自のストリームクラスに<<と>>をオーバーロードする場合も，次の節で述べるように，通常は同じようなコードを記述する必要があります。

ストリームクラスと TStreamable

ストリームクラスとは，ObjectWindows のストリームマネージャが提供するツールを使って，そのオブジェクトを寿命の長いストリームで読み書きできるクラスのことです。ストリームクラスは，適切な入出力演算子に加え，階層内に ***TStreamable*** を基本クラスとして持つ必要があります。なお，入出力演算子は通常，オーバーロード演算子<<と>>ですが，独自に定義することもできます。ObjectWindows クラスのツリー構造を見ればわかるように，***TWindowsObject*** は ***Object*** と ***TStreamable*** の両方から派生しています。したがって，***TWindowsObject*** から派生するすべてのクラスは ***TStreamable*** の「魔法」を継承しているわけです。また，***TWindowsObject*** から派生したすべてのクラスは，オーバーロード演算子<<と>>を持っているので，ストリームクラスです。***TScroller*** もまた ***TStreamable*** から多重継承しているので，ストリームクラスです。

ストリームマネージャ

複雑なオブジェクトをストリームに保存したり取り出したりするには，舞台裏の作業が必要です。この作業は裏方役のストリームマネージャが行ないます。データオブジェクトをストリームに書き込むと，バイナリイメージのまま格納したり取り出せるので簡単です。しかし，クラスオブジェクトの場合は，他の複雑なオブジェクトを指すポインタや *vtable*（仮想関数テーブル）を指すポインタなど，各種のデータ型を保持している場合があります。

このような厄介な問題に対処するため，ストリームマネージャは ***TStreamableTypes***，***TStreamable***，***TStreamableClass*** の各クラスを使ってストリームクラスのデータベースを管理しています。このオーバーヘッドはストリームクラス当たり約12バイトです。これらのクラスが組み合わされて，特定のオブジェクトに対する基本的な登録，読み込み，書き込み，構築の機能を行ないます。ストリームマネージャでクラスを登録する理由は，そのクラスをストリームクラスとしてストリームマネージャに認識させ，このデータベースに正しくエントリさせるためです。構築（ビルド）操作については次の節で取り上げます。

さらに，ストリームでの読み書きの間，ストリームマネージャはストリームで読み書きされるすべてのオブジェクトのデータベースを維持する必要があります。その操作は舞台裏の仕事なので深入りは避けますが，ポインタを使ってストリームにオブジェクトを書き込む場合を考えてみましょう。*ptr 1* と *ptr 2* は同じオブジェクトを指すポインタで，**ptr 1* と **ptr 2* の両方をストリームに書き込んだと仮定します。後でこれらのオブジェクトをストリームから読み出すと，同じオブジェクトのコピーが2つ，別々のポインタがついて現われ，混乱することになります。ストリームマネージャのデータベースではこのオブジェクトの書き込みの問題が起こりません。**ptr 1* のコピーのみがストリームに書き込まれるからです。同様に，オブジェクトをストリームから**ptr 1* と**ptr 2* に読み出すときも，オブジェクトは1つだけ作成され，*ptr 1* と *ptr 2* はこのオブジェクトを指しています。

ストリームマネージャはストリームオブジェクトとも密接に連動し，正しい ***read*** 関数と ***write*** 関数を提供します。***TStreamable*** には純粋仮想関数 ***read***（リーダ）と ***write***（ライタ）があります。次のストリームクラスから派生した2つの関数は，再定義が必要です。

```
class TStreamable {
:
protected:
  virtual Pvoid read(Ripstream) = 0;
  virtual void write(Ropstream) = 0;
};
```

ライタの役目は，オブジェクトの必要なフィールドをすべてストリームに書き込むことです。各ストリームクラスにライタが必要ですが，その実現は基本クラスのライタを呼び出せば，大幅に簡略化できます。***TWindow*** :: ***write*** の定義を以下に示します。

```
void TWindow::write(Ropstream os)
{
  long SaveStyle;
  BOOL NameIsNumeric;
  TWindowsObject::write(os);
  if ( !IsFlagSet(WB_FROMRESOURCE) )
  {
    SaveStyle = Attr.Style & ~(WS_MINIMIZE | WS_MAXIMIZE);
    if ( HWindow )
      if ( IsIconic(HWindow) )
        SaveStyle |= WS_MINIMIZE;
      else
        if ( IsZoomed(HWindow) )
          SaveStyle |= WS_MAXIMIZE;
    os << SaveStyle << Attr.ExStyle << Attr.X
       << Attr.Y << Attr.W << Attr.H
       << (long)(Attr.Param);
  }
  os << Attr.Id;

  NameIsNumeric = HIWORD(Attr.Menu) == NULL;
  os << NameIsNumeric;
  if ( NameIsNumeric )
    os << (long)(Attr.Menu);
  else
    os.fwriteString(Attr.Menu);

  os << Scroller;
}
```

リーダ（読み込み関数）の動作はライタの動作に似ています。各ストリームクラスで ***TStreamable*** の純粋仮想読み込み関数を定義し直す必要がありますが，これはたいてい基本クラスのリーダを付加フィールドまで拡張させて解決しています。

読み込み条件によっては，特殊な問題が発生することもあります。たとえば，ストリームのオブジェクトを該当する型の既存オブジェクトに読み込むと，読み込み動作では該当するデータと vtable ポインタを転送するだけです。しかし，読み出すオブジェクトが存在しなければ，ストリームから初期化できる一種の骨組みだけのオブジェクトを構築しておく必要があります。この仕事を受け持つのが**ビルダ**です。ビルダは専用の ***StreamableInit***，すなわちビルドコンストラクタを呼び出してオブジェクトを保持できる大きさのメモリを確保し，その vtable を構築します。このような読み込み条件に遭遇しうる各ストリームクラスには，ビルダとビルドコンストラクタが必要です。

リーダ，ライタ，ビルダ，ビルドコンストラクタは，ObjectWindows の標準ストリームクラスに定義済みです。独自のストリームクラスを作成するときは，必ず用意してください。

ストリームクラスのコンストラクタ

すでに述べたように，ストリームからの読み込み操作の対象となるオブジェクトを持つストリームクラスには，専用のコンストラクタが必要です。このコンストラクタは，OBJSTRM.H で定義されている enum 定数の引数 *streamableInit* だけをとります。同様に，標準のストリームクラスにもすべてこのようなコンストラクタがあります。これは public としてください。この形式を以下に示します。

```
class TMyStreamable : public TBase, public TStreamable {
:
public:
  TMyStreamable(StreamableInit s);
:
};
```

データ型 ***StreamableInit*** は *streamableInit* だけをメンバとして持つ enum です。このコンストラクタでオブジェクトを作成すると，標準コンストラクタの場合とは異なり，含まれているポインタのコンストラクタを呼び出しません。

```
TMyStreamble str(streamableInit);
```

したがって，メモリ管理が簡単です。ストリームのオブジェクトをオブジェクト *str* に読み込む場合，オブジェクトのリーダは *str* 内の不要データを解放する必要がありません。

```
TMyStreamable str(streamableInit);
// 空の文字列を作成
:
ifpstream ifps("str.sav");     // 入力用ファイルストリームをオープン
ifps >> str;                   // オブジェクトを文字列に読み込む
:
```

なお，***streamableInit*** コンストラクタで作成した *str* などのオブジェクトは，ストリーム読み込み操作で「初期化」するまでは安心して使えません。enum とメンバ名が無意味になる場合があるからです。この場合，このコンストラクタを他から区別するための固有のデータ型引数しか与えられません。

ストリームクラスのメンバ関数 ***build*** は次のように定義されます。

```
PTStreamable TMyStreamableClass::build()
{
  return new TMyStreamableClass(streamableInit);
}

TMyStreamableClass::TMyStreamableClass(StreamableInit s) :
  TBaseClass(streamableInit)
{
}
```

つまり，ビルドコンストラクタは基本クラスのコンストラクタを呼び出すだけです。この点は一貫して変わりません。

ストリームクラス名

各ストリームクラスで，***TStreamable*** から派生させたプライベート仮想関数 ***streamableName*** をオーバーライドする必要があります。この関数はクラスの固有名をヌルで終わる文字列として返さなければなりません。

```
class TMyStreamable : public TBase, public TStreamable {
:
private:
  virtual const Pchar streamableName() const
  { return "TMyStreamable"; }
};
```

上の処理も ObjectWindows の標準ストリームクラスに対してはすでに済んでいます。ただし，独自のストリームクラスを定義する場合は，個々にオーバーライドが必要です。通常，上のように ***streamableName*** にクラス名を返させます。ストリームマネージャは，管理している各種のクラスデータベースでこの固有名を使います。

ストリームマネージャの使用

ストリームマネージャを使うときは次の手順で行ないます。

- ストリームマネージャコードでリンクする。
- 適切なストリームマネージャを作成する。
- ストリームオブジェクトを使用する。

次に各手順を詳しく説明します。

ストリームマネージャコードでリンクする

どのストリームクラスであってもオブジェクトを操作する場合は ***read***, ***write***, ***build*** の3つの関数を定義する必要があります。これらの関数をストリームマネージャに認識させ，ストリームマネージャを使うあらゆるアプリケーションにリンクしなければなりません。このリンクとはクラス名をストリームマネージャに登録することです。リンクには ***__link*** マクロを使います。このマクロの目的は，***TStreamableClass*** 型のオブジェクトへの参照を提供することです。規則上，このオブジェクト名は **Reg*classname*** とします。クラス名の最初に T がある場合は除きます。次のように BASE.CPP で定義されている ***TBase*** というクラスがあり，

```
class TBase : public TStreamable
{
  int x;
  int y;
};
```

また，MYAPP.CPP で定義されている *TDerived* に *TBase* オブジェクトを指すポインタのデータメンバがある場合，*TBase* のストリーム機能をリンクするように__***link*** マクロでストリームマネージャに伝えてください。ストリームクラスの登録と *TDerived* 定義を以下に示します。

```
__link(RegBase)
class TDerived : public TStreamable
{
  int z;
  PTBase zz;
};
```

P - 型はポインタです。

☞ データメンバが，ソースファイル外で定義されたクラスを指すポインタの場合は，必ず__***link*** マクロを使ってください。

ストリームオブジェクトの作成

ストリームオブジェクト ***ipstream*** または ***opstream*** を作成するには，該当する引数で宣言しなければなりません。これは，**iostream** の場合とまったく同様です。DLG.SAV というファイルストリームにダイアログを保管するには，次のように ***ofpstream*** オブジェクトを宣言してください。

各種のストリームコンストラクタと引数については，「ObjectWindows for C++リファレンスガイド」の第2章『ストリームクラス』を参照してください。

```
TChDirDialog cdlg;
:
// ここでダイアログを作成

ofpstream of("dlg.sav");
// 出力ファイルストリームをオープン

of << cdlg;
// ダイアログオブジェクトをファイルストリームに書き込む
```

これで，次のコンストラクタがデフォルトで呼び出されます。

```
ofpstream(PCchar filename, int mode = ios::out,
          int prot = filebuf::openprot)
```

ストリームオブジェクトの使用

代表的な読み込み操作を以下に示します。

```
TChDirDialog cdlg (streamableInit);
// ビルドコンストラクタを呼び出し、オブジェクトの骨組みを作成

ifpstream ifps("dlg.sav");
// 入力ファイルストリームをオープン

ifps >> cdlg;
// ファイルストリームのダイアログオブジェクトをcdlgに読み込む
```

ストリームにおけるコレクション

C++のコンテナクラスは，デフォルトではストリームクラスではありません。次の節で，既存のコンテナクラスから新しいコンテナクラスを派生させてストリームクラスに変える例を示します。順を追ってクラスをストリームクラスに変える説明をしながら，実現に必要な各必須要素をストリームマネージャがどのように使うのかも解説します。次の例はEXAMPLES¥STEPSディレクトリにあるSTEP 9.CPPです。

配列をストリームクラスにする

プログラム例では，コンテナクラスライブラリの **Array** クラスを使って，*TPoint* のコレクションを格納しています。このコレクションはこのプログラム例で定義されているオブジェクトです。**Array** クラスそのものはストリームクラスではないので，新しいクラス *TPointArray* を定義します。*TPointArray* は **Array** と ***TStreamable*** の両方から多重継承しています。すべてのストリームクラスは ***TStreamable*** から継承しなければなりません。

ストリームクラスのビルダ関数

ストリームクラスでは必ず，スタティックメンバ関数 ***build***，専用コンストラクタ，および ***read***，***write***，***streamableName*** の 3 つの仮想関数を定義してください。これらのメンバ関数のそれぞれに対する宣言は以下のクラス定義で行ないます。

```
class TPointArray : public Array, public TStreamable {

public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0) :
    Array(upper, lower, aDelta) {};
⋮
  public:
    static PTStreamable build();
  protected:
    TPointArray(StreamableInit) : Array(50, 0, 50) {};
    virtual void write(Ropstream);
    virtual Pvoid read(Ripstream);

private:
  const Pchar streamableName() const {
    return "TPointArray"; }
};
```

次のコードは *TPointArray* をストリームに書き込みます。

```
ofpstream os("save.dat");
PTPointArray APointArray;
⋮
os << APointArray;
```

多くの場合，必須というわけではありませんが，各ストリームクラスに以下のように**挿入子**と**抽出子**を定義しておいてください。この 2 つはオブジェクトとオブジェクトの ***TStreamable*** 構成要素の間で型変換を行なうものです。

```
inline Ripstream operator >> (Ripstream is, RTPointArray cl)
  { return is >> (RTStreamable)cl; }

inline Ripstream operator >> (Ripstream is, RPTPointArray cl)
  { return is >> (RPvoid)cl; }

inline Ropstream operator << (Ropstream os, RTPointArray cl)
  { return os << (RTStreamable)cl; }

inline Ropstream operator << (Ropstream os, PTPointArray cl)
  { return os << (PTStreamable)cl; }
```

このコード例では，*TPointArray* を指すポインタがストリーム *os* に書き込まれます。また，4 番目のインライン定義が呼び出され，*PTStreamable* を引数とする挿入子（演算子<<）を呼び出します。この挿入子はストリームの実現によって定義され，引数として指定されたオブジェクトの仮想メンバ関数 ***write*** を呼び出します。次は，*TPointArray* :: *write* を呼び出す例です。

```
void TPointArray::write(Ropstream os) {
  RContainerIterator PointIterator = initIterator();
  os << getItemsInContainer();

  while (int(PointIterator) != 0) {
    RObject PointObject = PointIterator++;
    if (PointObject != NOOBJECT)
    {
      os << ((PTPoint)(&PointObject))->X;
      os << ((PTPoint)(&PointObject))->Y;
    }
  }

  delete &PointIterator;
}
```

TPointArray :: *write* は，**Array** コンテナに繰り返し処理を割り当て，配列内の要素の数をストリームに書き込み，次に配列に対して繰り返し処理を実行し，配列の各要素を書き出し，最後に繰り返し処理オブジェクトを破棄します。この一連の処理をさらにモジュール化するには，*TPoint* クラスをストリームクラスにすればよいということはすでに述べました。この方法によると，ループで単に *TPoint* オブジェクトをストリームに書き込むだけでよく，*TPoint* のデータメンバ（***X*** と ***Y***）をいちいち書き出す必要はありません。

streamableName メンバ関数

ストリームにオブジェクトを書き込むと，ストリームマネージャはオブジェクトの仮想メンバ関数 ***streamableName*** を呼び出して，オブジェクトの名前を決めます。この名前をストリームに書き込んでからオブジェクトが書き出されます。インラインメンバ関数 ***streamableName*** の定義をクラス定義から以下に示します。

```
class TPointArray : public Array, public TStreamable {
:
private:
  const Pchar streamableName() const
    { return "TPointArray"; }
:
};
```

ストリームからオブジェクトを読み出すとき，ストリームマネージャは最初にクラス名を示す文字列を読み込みます。この名前は ***TStreamableClass*** のスタティックコンストラクタを呼び出してストリームマネージャで登録しておいたものです。すべてのストリームクラスに ***TStreamableClass*** 型のスタティックインスタンスを作成し，次の登録をさせてください。

```
TStreamableClass RegPointArray("TPointArray", TPointArray::build,
                               __DELTA(TPointArray));
```

このコンストラクタは，クラス名 *TPointArray*，ビルダ関数(*TPointArry* :: *build*)のアドレス，オブジェクトの基底からオブジェクトの ***TStreamable*** 構成要素までのオフセットを指定します。このオフセットは ***__DELTA*** マクロが自動的に計算します。

ストリームマネージャは，スタティックオブジェクトを使って，すべてのストリームクラスのデータベースを管理します。*TPointArray* 型のオブジェクトをストリームから読み出すと，このデータベースを検索してビルダのスタティックメンバ関数 *TPointArray* :: *build* のアドレスを見つけます。

```
PTStreamable TPointArray::build()
{
  return new TPointArray(streamableInit);
}
```

ビルダ関数はいずれも ***StreamableInit*** 型を引数とする専用のコンストラクタを呼び出すだけです。このコンストラクタは基本クラスのコンストラクタを呼び出しますが，データメンバは初期化しません。したがって，作成されたオブジェクトは正しいサイズになり，その仮想テーブルも埋まります。

コンストラクタの定義もクラス定義の中で行ないました。***StreamableInit*** 型は，1つの値 *streamableInit* だけを持つ enum です。ストリーム化用の専用コンストラクタは，enum 引数によってクラスのすべての対象コンストラクタから選ばれます。

ストリームクラスのリーダ関数

利用できるオブジェクトがあるとき，ストリームマネージャは仮想メンバ関数 ***read*** を呼び出すことができます。以下にこの定義を示します。

```
Pvoid TPointArray::read(Ripstream is) {
  sizeType NumPoints;
  PTPoint APoint;

  is >> NumPoints;

  for (int i = 0; i < NumPoints; ++i)
  {
    APoint = new TPoint(0, 0);
    is >> APoint->X;
    is >> APoint->Y;
    add(* (APoint));
  };

  return this;
}
```

この例では，*TPointArray* のリーダ関数が配列内の要素数を読み込み，次に各 *TPoint* に対応する X 値と Y 値のペアを読み込みます。さらに，**Array** を指す新しい *TPoint* を追加します。

前にも述べたように，*TPoint* 自体をストリーム形式にすれば，処理が簡単になります。ストリーム形式にして，*TPoint* のストリーム機能を別のファイルに実現してあれば，リンカに指示して *TPoint* オブジェクトのストリーム機能をリンクさせることができます。***TStreamableClass*** 登録オブジェクトが ***RegPoint*** という名前の場合，次のようにすれば，ストリーム機能を強制的に ***__link*** マクロを使ってリンクできます。

```
__link(RegPoint)
```

これで，***TStreamableClass*** 型のオブジェクト *RegPoint* への外部参照を行なうことができます。この登録オブジェクトにはスタティックメンバのビルダ関数を指すポインタがあるので，関数がリンクされます。その結果，専用のストリームコンストラクタと仮想リーダ関数，仮想ライタ関数がリンクされます。

第3部

Windows プログラミング

第16章

メモリ管理

通常は，Windowsのメモリ管理機構を直接操作する必要はありません。ほとんどの場合，Borland C++がメモリ管理を代行します。普通は動的変数を割り当てる標準手続き **new** と **delete**（あるいは **malloc** と **free**）ルーチンを用いるだけで，コンパイラが適切なメモリ領域をWindowsのヒープから割り当ててくれます。

しかし，メモリ管理機構を直接操作する必要が生じる場合もあります。以下の節では，このような場合のWindowsメモリの割り当てや使用方法についておおまかに説明します。

メモリマネージャの使い方

Windowsのアプリケーションを開発する場合，コンパイルしたプログラムで直接使用可能なメモリ容量以上のメモリを必要とする場合があります。たとえば，アプリケーションにディスクからビットマップファイルを読み込んで，画面に表示する機能があるとします。このようなアプリケーションプログラムでは，Windowsにビットマップデータを渡して表示する前に，ビットマップデータをどこかに格納する必要があります。ビットマップデータは非常に大きいこともあります。この種のデータの格納場所をメモリ内に確保できるように，Windowsでは本来プログラムが扱えるメモリ領域以外のメモリの割り当て，操作をするための関数を提供しています。

Windowsはマルチタスクシステムなので，複数のアプリケーションでメモリを共用できます。Windowsは，各アプリケーションが最も効率よくメモリを使えるようにメモリ管理を行ないます。この章では，Windowsで使えるメモリの種類と，アプリケーション内からWindowsのメモリを確保し，アクセスする際に知っておくべき事柄について説明します。

Windowsのメモリ管理

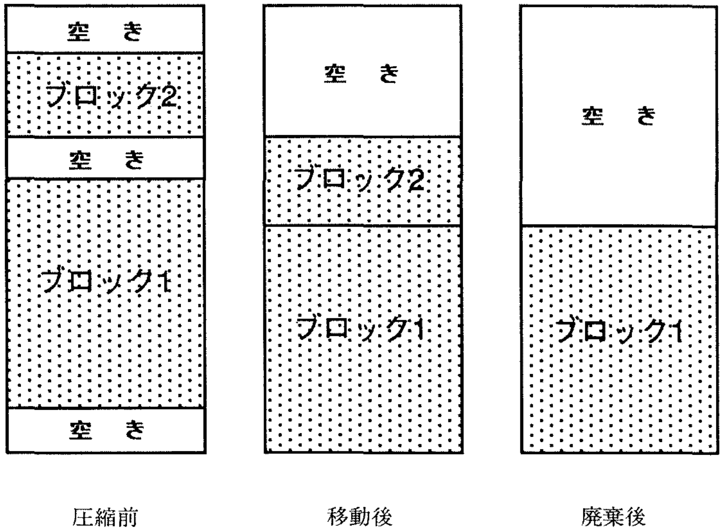
DOS上の大半のメモリ管理システムでは，確保されたメモリ領域は使用中は特定の位置に固定されたままになっています。一方，Windowsではメモリ領域を**移動**したり，**廃棄**したりすることができるようになっています。メモリブロックを移動可能属性をつけて確保すると，メモリブロックは固定アドレスを持たなくなります。Windowsは利用可能なメモリを効率よく運用するために，必要ならその領域を移動させます。たとえば，離れた位置に割り当てられている2つのメモリブロックを移動して，連続した1つのブロックに統合し，より大きな空メモリブロックを確保したりします。

Windowsでは，メモリを確保する際に廃棄可能属性をつけることもできます。廃棄可能メモリは，メモリブロックを移動できるという点では移動可能メモリと同じですが，ブロック長を0にして再割り当てすることにより，メモリを廃棄できるという点が異なります。ただし，この方法で再割り当てをすると，メモリブロック内にあったデータはすべて破壊されます。したがって，アプリケーション側でメモリを再確保して，正しいデータをそのメモリブロックに再ロードしなければなりません。メモリブロックを移動して，連続したより大きな空きメモリを確保することを**圧縮**と呼んでいます。

図16.1の3つの図に圧縮過程の一例を示します。左の最初の図では，3つの空きメモリブロックがメモリブロック1とメモリブロック2によって分断されています。この3つの空きメモリブロックのどれよりも大きなサイズのメモリを確保したいという要求があったとします。メモリブロック1とメモリブロック2がともに移動可能であれば，圧縮によってこの2つのブロックを移動して，連続したブロックに統合し，より大きな空きメモリ領域が生成されます（中央の図)。さらにこの圧縮によっても割り当てに充分な空きメモリ領域が得られない場合を考えてみます。もしメモリブロック2が廃棄可能であれば，メモリからこのブロックを除去し（大きさ0で再割り当てする)，より大きな連続した空きメモリを確保

します（下の図）。

図 16.1
圧縮の例



ハンドルとアドレス

以上の説明だけでは，アプリケーション側でWindowsのメモリを管理するのは難しいように思われますが，幸いなことにそうでもありません。アプリケーションがWindowsのメモリを確保すると，確保されたメモリブロックへのポインタではなく，ハンドルが返されるからです。このハンドルはメモリ上の物理的な位置を示すアドレスではなく，メモリブロックを参照する識別子です。アプリケーションではこのメモリを操作する関数を後で呼び出すとき，この識別子でメモリブロックを参照します。値が0のハンドルは無効となります。

しかし，移動可能なメモリブロック内のデータを利用するために，実際のメモリアドレスが必要になる場合があります。メモリブロックの実際のアドレスを得るには，そのブロックをロックしてください。メモリブロックをロックすると，そのメモリブロックの先頭のアドレスが得られます。メモリブロックがロックされている間は，ロック操作によって得られたポインタはロックが解除されるまで有効となります。ロック中は，そのメモリブロックが移動可能あるいは廃棄可能であったとしてもポインタは有効です。

リアルモードの留意点

リアルモードでは，メモリブロック内のデータを使用し終わったら，ブロックのロックを解除してください。ロックを解除しないと，Windowsがメモリブロックを移動または廃棄して空きメモリを増やすことができなくなるため，メモリ操作の効率が下がります。メモリブロックのロックを解除すると，ロック操作で得られたポインタは無効となります。

プロテクトモードの留意点

Windowsのプロテクトモード（スタンダードモードか386エンハンスモード）では，アプリケーションの使用中にグローバルメモリブロックを確保したり，ロックしてもなんら問題はなく，効率も性能も落ちません。

属性を移動可能としてメモリブロックを確保してみてください。Windowsが必要に応じてこの物理メモリを移動させたとしても，もとのアドレスを使ってメモリを使えることがわかるはずです。

ローカルメモリとグローバルメモリ

メモリブロックは，ローカルヒープまたはグローバルヒープのどちらからでも確保して使用することができます。Windowsのシステム，Windows上で動作するアプリケーション，Windowsアプリケーションによって確保されたグローバルメモリブロックなどは，すべてグローバルヒープ上に配置されます。ローカルヒープは，1つのアプリケーション内でしか利用できないメモリのことをいいます。

要求されるメモリの性質に応じて，以下の点を考慮した上でローカルヒープかグローバルヒープのどちらからメモリを確保するかを決めることになります。

ローカルメモリの特性：

- アクセスが速い。
- サイズが限られている（64キロバイト未満）。

グローバルメモリの特性：

- 32バイトごとに区切られたメモリブロックである。
- 1ブロックに対し20バイトのWindows予約領域が取られるため，オーバーヘッドがある。
- 全アプリケーションで扱える合計ハンドル数に限界がある。
- アクセスが遅い。
- より大きなメモリが利用可能。

どちらにするかは，通常は必要とするメモリ容量によって決まります。メモリが1Kバイト以上必要な場合は，グローバルメモリを利用する方が賢明です。一方，小さなメモリで，しかも短時間しか使用しない場合は，ローカルメモリを使う方がよいでしょう。

ローカルヒープの使い方

ローカルヒープ内のメモリは，アプリケーションの特定のインスタンスからしかアクセスできません。Windowsでは，ローカルヒープは通常アプリケーションのデータセグメント上に設定されます。このデータセグメントには他に，スタック，およびアプリケーションのスタティック変数も配置されています。64 KBまでのデータセグメントの残りの領域が，すべてローカルヒープとして利用可能です。

Windowsが自動的にアプリケーションにローカルヒープを設定するわけではないので，リンク時にHEAPSIZE文をモジュール定義（.DEF）ファイルに入れてローカルヒープの大きさを指定しなければなりません。TLINKのデフォルトのヒープサイズである4096バイトをそのまま使うこともできます。

ローカルヒープの最大サイズは，ローカルヒープを含むデータセグメントが固定か移動可能かによっても異なります。固定の場合は，確保できるローカルメモリブロックの最大サイズはコンパイルとリンク時に指定したヒープサイズになります。データセグメントが移動可能で，ローカルヒープサイズとして指定した値よりも大きなメモリ量を要求された場合，さらにメモリを確保して（最大64 KB）要求を満たそうとします。必要なら，Windowsはメモリをさらに追加できるようにデータセグメントを移動させます。この場合，ローカルデータを指すfarポインタはすべて無効となります。

ローカルメモリの確保と利用

LocalAlloc 関数を呼び出すと，引数として指定した大きさと種類のメモリブロックを確保します。この際，確保するメモリブロックを固定とするか，移動可能にするか，移動可能でかつ廃棄可能（廃棄可能とするには，移動可能でなければならない）にするかを指定します。関数呼び出しの *Flags* 引数で指定するオプションは以下のとおりです。

- 固定メモリブロックを指定する場合は ***LMEM_FIXED***。
- 移動可能なメモリブロックを指定する場合は ***LMEM_MOVEABLE***。
- 廃棄可能メモリブロックを指定する場合は ***LMEM_MOVEABLE | LMEM_DISCARDABLE***。

ローカルヒープから新規にブロックを確保すると，ローカルヒープ内の他のブロックが移動または廃棄される場合があります。他のブロックが廃棄されないようにするには，引数 *Flags* に ***LMEM_NODISCARD*** を追加します。他のブロックが移動も廃棄もされないようにするには ***LMEM_NOCOMPACT*** を追加します。

LocalAlloc 関数は，メモリ確保に成功するとそのメモリブロックに対するハンドルを返し，確保に失敗すると 0 を返します。成功した場合，要求したサイズ（バイト単位）は必ず確保されますが，要求したサイズよりも大きくなることもあります。実際の大きさを得るには，***LocalSize*** 関数を使用します。ローカルブロックとして確保できる最大サイズは，ローカルヒープのサイズにより決まります。

ブロックに対するハンドルを得たからといって，それでブロックのデータにアクセスできるわけではありません。アクセスするには，***LocalLock*** 関数を呼び出してブロックをロックし，メモリ内でのアドレスを得ます。***LocalLock*** 関数は，一時的にメモリブロックをローカルヒープの特定の場所に固定し，そのブロックへのポインタを返します。この関数によって返されるアドレスは，***LocalUnlock*** 関数によってメモリブロックのロックが解除されるまで有効です。***LocalLock*** 関数はブロックのロック回数を 1 回分増加させます。このロック回数はリファレンスカウントとも呼ばれ，メモリブロックがメモリ内にロックされた回数を記録したものです。

☞ ***LocalLock*** 関数の戻り値は，常に正しい値が得られるとは限らないので必ずチェックしなくてはなりません。渡したハンドルが無効だった場合やブロックが廃棄されていた場合（ブロックが ***LMEM_DISCARDABLE*** と指定された場合）は，戻り値がヌルになります。以下に256バイトのローカルメモリブロックの正しい確保方法を示します。

```
HANDLE TheLocalHandle;
PSTR TheLocalData;
:
TheLocalHandle = localAlloc(LMEM_MOVEABLE | LMEM_DISCARDABLE, 256);
if (TheLocalHandle != 0) {
  TheLocalData = LocalLock(TheLocalHandle);
  if (TheLocalData != NULL)
  {
    // TheLocalData をポインタとして用いてデータを処理
    LocalUnlock(TheLocalHandle);
  }
  else // ロック失敗；
}
else // メモリ確保失敗；
}
```

LMEM_FIXED 属性で確保したローカルメモリブロックは，メモリ内を移動することはありません。そのため，こういったブロックを固定メモリにロックするために ***LocalLock*** 関数を呼ぶ必要はありません。この場合に限り，***LocalAlloc*** 関数呼び出しで返される16ビットのハンドルは，メモリブロックの16ビットアドレスそのものとなります。

ローカルメモリブロックの解放と廃棄

LocalFree 関数を使えば，ローカルメモリブロックを解放することができます。ローカルブロックを解放すると，ブロックはローカルヒープの空きスペースに戻り，そのハンドルが無効になります。アプリケーションでは，常に終了前に全メモリブロックを解放すべきです。また，どんなメモリブロックでも必要なくなった時点ですぐに解放するようにした方がよいでしょう。

LocalDiscard 関数を用いて，明示的にローカルメモリブロックを廃棄することができます。ローカルメモリブロックを廃棄すると，メモリブロックの大きさがゼロになるよう再割り当てが行なわれ，ブロックの内容がローカルメモリから削除されます。しかし，ハンドルはこの後も有効で，このハンドルと新しいサイズを使って ***LocalReAlloc*** 関数を呼び出せば再び使用できます。ハンドルの廃棄と再確保を使えば，メモリブロックを解放して新たに確保するよりもアプリケーションの処理時間を節約することができます。

```
HANDLE LocalHandle;
:
// TheLocalHandle は割り当ててある
LocalDiscard(TheLocalHandle);
// このハンドルを再利用して新しいメモリブロックを作成
LocalHandle = LocalReAlloc(TheLocalHandle, 256, LMEM_MOVEABLE);
```

ローカルメモリブロックは，ロック回数がゼロでない限り，解放したり廃棄したりすることはできません。

ローカルメモリブロックの再確保と変更

LocalReAlloc 関数を使えば，メモリブロックの内容を保持したままブロックの大きさを変更できます。Windows は，現行サイズよりも小さいサイズを指定すると，ブロックを切り詰めます。現行サイズよりも大きいサイズを新しく指定すると，新しい領域は ***LMEM_ZEROINIT*** が指定されていればゼロで初期化されますが，指定されていなければ，ブロックのデータは不定となります。***LocalAlloc*** 関数の場合と同様，***LocalReAlloc*** 関数によってメモリブロックが再確保される際には，Windows によって既存のローカルメモリブロックの移動や廃棄が行なわれることもあります。***LMEM_NODISCARD*** や ***LMEM_NOCOMPACT*** を指定することによって，再確保中に他のブロックが廃棄されたり，移動されたりしないようにすることもできます。

LocalReAlloc 関数を使って，***LMEM_MOVEABLE*** や ***LMEM_DISCARDABLE*** などのメモリブロックの属性を変更することができます。この場合，新しく設定する属性に加えて，***LMEM_MODIFY*** を指定する必要があります。***LMEM_MODIFY*** 属性を指定して ***LocalReAlloc*** 関数を呼び出しても，メモリブロックから ***LMEM_FIXED*** 属性を取り去ったり，***LMEM_FIXED*** 属性を付加したりすることはできません。以下に示すコードでは，ブロックを廃

棄可能に変更しています。

```
HANDLE TheLocalHandle;
:
TheLocalHandle = LocalAlloc(LMEM_MOVEABLE, 256);
TheLocalHandle = LocalReAlloc(TheLocalHandle, 256, LMEM_MODIFY |
                                LMEM_DISCARDABLE);
```

ローカルメモリブロック情報の取得

LocalSize 関数や ***LocalFlags*** 関数を用いてローカルメモリブロックに関する情報を取得することができます。***LocalSize*** 関数は，ローカルブロックの大きさを返します。メモリ確保を行なうと要求した大きさ以上のブロックが生成される場合があるので，必要なら ***LocalSize*** 関数を用いてブロックの実際の大きさを調べてください。

```
HANDLE TheLocalHandle;
unsigned int BlockSize;

TheLocalHandle = LocalAlloc(LMEM_MOVEABLE, 256);
if (TheLocalHandle != 0)
  BlockSize = LocalSize(TheLocalHandle);
```

LocalFlags 関数を使えば，メモリブロックのロック回数や，そのメモリブロックが廃棄可能か，廃棄可能ならばブロックがすでに廃棄されたかなどを調べることができます。

```
HANDLE TheLocalHandle;
unsigned int Flags;
unsigned int LockCount;

// ブロックはすでに確保されているものとする
Flags = LocalFlags(TheLocalHandle);
LockCount = Flags & LMEM_LOCKCOUNT;
```

LocalCompact 関数は，単に引数として 0 を指定すれば，ローカルヒープから取れるメモリブロックの最大サイズを返します。引数として 0 以外の値を指定すれば，必要に応じてメモリの圧縮と廃棄を行ない，ローカルヒープ内の連続した空きメモリ領域を確保します。

プログラミング上の留意点

移動可能でかつ廃棄可能なメモリブロックへのアクセスが終了したら，すみやかにロックを解除してください。メモリブロックのロックを解除しないと，それ以降のメモリ確保のためにブロックを移動して必要な空きメモリを生成することができません。また，プログラムが終了する前にメモリブロックをすべて解放してください。

グローバルヒープの使い方

グローバルヒープとは，Windowsとアプリケーションで共有するシステム全体のメモリ領域のことです。アプリケーションで利用できるグローバルヒープの容量は，Windowsの動作モードによります。グローバルメモリは共有領域なので，アプリケーション側で効率よく使用しないと，システム全体の効率が下がります。

グローバルメモリの確保と利用

GlobalAlloc 関数は，引数に指定された大きさとクラスのメモリブロックを確保します。この際，確保するメモリブロックを固定とするか，移動可能にするか,移動可能でかつ廃棄可能(廃棄可能とするには移動可能でなければならない)にするかを選択しなければならないことは，ローカルメモリの場合と同じです。この選択は，関数呼び出しの引数 *Flags* で以下のように指定します。

- 固定メモリブロックを指定する場合は ***GMEM_FIXED***。
- 移動可能なメモリブロックを指定する場合は ***GMEM_MOVEABLE***。
- 廃棄可能メモリブロックを指定する場合は ***GMEM_MOVEABLE | GMEM_DISCARDABLE***。

グローバルヒープから新規にブロックを確保すると，グローバルヒープ内の他のブロックの移動や廃棄が起こる場合があります。他のブロックが廃棄されないようにするには，引数 *Flags* に ***GMEM_NODISCARD*** を設定します。他のブロックが移動も廃棄もされないようにするには，引数 *Flags* に ***GMEM_NOCOMPACT*** を設定します。

GlobalAlloc 関数は，メモリ確保に成功するとそのメモリブロックに対するハンドルを返し，確保に失敗すると 0 を返します。***GlobalAlloc*** 関数で返される値を常にチェックして，メモリ確保に成功したかどうかを確認してください。メモリ確保に成功すると，要求したサイズは必ず確保されますが，要求したサイズよりも大きくなる場合もあります。実際の大きさを得るには ***GlobalSize*** 関数を使用します（280 ページ参照）。グローバルメモリブロックとして確保できるサイズは，Windows がスタンダードモードで動作している場合は 1 MB まで，386 エンハンスモードで動作している場合は，64 MB までです。

グローバルメモリブロックのアドレスを得るには，***GlobalLock*** 関数を呼び出してブロックをロックします。リアルモードでは，***GlobalLock*** 関数は一時的にブロックを物理メモリの固定領域に固定します。プロテクトモードではブロックは固定されませんが，ブロックがメモリ内を移動してもアドレス（セレクタ）は有効です。どのモードでも ***GlobalLock*** 関数は far ポインタを返します。この far ポインタは，***GlobalUnlock*** 関数でロックが解除されるまで有効です。

GlobalLock 関数で返される値は，必ずしも正しいポインタとは限らないので，必ずチェックしてください。渡したハンドルが無効だった場合やブロックが廃棄されていた場合は，戻り値がヌルになります。以下にグローバルメモリブロックを確保するコード例を示します。

```
unsigned short BigArray[20000];
unsigned short far * TheGlobalData;
HANDLE TheGlobalHandle;

TheGlobalHandle = GlobalAlloc(GMEM_MOVEABLE, sizeof(BigArray));
if (TheGlobalHandle != 0)
{
  TheGlobalData = GlobalLock(TheGlobalHandle);
  if (TheGlobalData != NULL)
  {
    // TheGlobalData をポインタとしてデータを処理
    GlobalUnlock(TheGlobalHandle);
  }
  else
  // ロック失敗:
}
else
// メモリ確保失敗:
```

ローカルメモリブロックとは異なり，グローバルメモリを確保する際に ***GMEM_FIXED*** 属性をつけた場合でも，データをアクセスする前にメモリブロックをロックする必要があります。固定ブロックに対して ***GlobalAlloc*** 関数が返すハンドルはこの場合のブロックへのポインタではありません。

グローバルメモリブロックをロックする別の方法

グローバルメモリをロックしたり，ロックを解除したりする関数は他にもいくつかあります。***GlobalFix*** 関数は，Windows の動作モードに関係なく，メモリ内でメモリブロックをロックしてブロックの移動を防ぎます。ブロックが実際に固定されるので，ロック回数が 1 回増加します。この関数は，Windows のメモリ圧縮機構を妨害する可能性があるので，本当に必要な場合しか使わないでください。ほとんどのアプリケーションはこの方法でメモリブロックを固定する必要はありません。

GlobalUnfix 関数は，ブロックのロックを解除し，ロック回数を 1 回減らします。

GlobalWire 関数はブロックをできるだけ低い物理アドレス位置で固定してロックし，ロック回数を 1 回増やします。***GlobalWire*** 関数は，長時間のロックが必要なメモリブロックに対して便利です。メモリブロックが低メモリに置か

れるので，固定されていても Windows が行なうメモリ圧縮を妨げないからです。***GlobalUnWire*** 関数は，ブロックのロックを解除してロック回数を1回減らします。

グローバルメモリブロックの解放と廃棄

ローカルメモリブロックと同様，グローバルメモリも解放，廃棄できます。違う点は関数名だけで，それぞれ ***GlobalFree*** と ***GlobalDiscard*** となります。

グローバルメモリブロックの再確保と変更

GlobalReAlloc 関数を使えば，メモリブロックの内容を保持しながらブロックの大きさを変更できます。現在の大きさより小さいサイズを指定すると，ブロックが切り詰められます。現在の大きさより大きいサイズを指定すると，新しい領域は ***GMEM_ZEROINIT*** が指定されていればゼロで初期化されますが，指定されていなければブロック内容は不定となります。***GlobalAlloc*** 関数の場合と同様，***GlobalReAlloc*** 関数によってメモリブロックが再確保される際には，既存のグローバルメモリブロックの移動や廃棄が行なわれることもあります。***GMEM_NODISCARD*** や ***GMEM_NOCOMPACT*** を指定することによって，再確保中に他のブロックが廃棄されたり，移動されたりしないようにすることもできます。

GlobalReAlloc 関数を使って，メモリブロックに移動可能属性や廃棄可能属性を設定できます。

メモリブロックを再確保する際に，64 KB の倍数を超えるような大きさを新しく指定する場合，ブロックに対して新しいハンドルが返される場合があります。スタンダードモード動作のアプリケーションでは，65,519 の倍数を超えるブロックの再確保の際に発生します。このため，もとのハンドルを保存しておいて，***GlobalReAlloc*** 関数が返した新しいハンドルをチェックしなければなりません。***GlobalReAlloc*** 関数の呼び出しが成功した場合は，もとのハンドルのかわりに新しいハンドルを使います。成功しなかった場合はもとのハンドルの値をそのまま使用します。以下に例を示します。

```
HANDLE TheGlobalHandle;
HANDLE OriginalHandle;
// ブロックは最初32KBとして確保
// 64KB境界を超えて再確保
OriginalHandle = TheGlobalHandle;
TheGlobalHandle = GlobalReAlloc(TheGlobalHandle, 102400,
                                GMEM_MOVEABLE);
if (TheGlobalHandle == 0)
  TheGlobalHandle = OriginaHandle;
```

グローバルメモリブロック情報の取得

GlobalSize 関数や ***GlobalFlags*** 関数を用いてグローバルメモリブロックに関する情報を取得することができます。***GlobalSize*** 関数は，グローバルブロックの大きさを返します。***GlobalFlags*** 関数を使えば，メモリブロックのロック回数や，そのメモリブロックが廃棄可能か，廃棄可能ならばブロックがすでに廃棄されたかなどを調べることができます。***GlobalFlags*** 関数はまた，ブロックが ***GMEM_DDESHARE*** と ***GMEM_NOT_BANKED*** のどちらを指定して確保されたのか示します。***GMEM_DDESHARE*** は，アプリケーションの複数のインスタンスと共有するメモリブロック，またはアプリケーションの複数のインスタンスが用いる DLL を共有するメモリブロックに使います。***GMEM_NOT_BANKED*** はリアルモードでのみ使います。

```
HANDLE TheGlobalHandle;
unsigned int Flags;
 :
Flags = GlobalFlags(TheGlobalHandle);
// ブロックが廃棄可能かどうかをテスト
if (Flags & GMEM_DISCARDABLE)
  // ブロックは廃棄可
else
  // ブロックは廃棄不可
```

GlobalCompact 関数では，引数として単にゼロを指定すれば，グローバルヒープから取れるメモリブロックの最大サイズを得ることができます。***GlobalCompact*** は常に最初にメモリを圧縮します。また，***GlobalCompact*** 関数に引数として -1 を渡すと，ロックされていない廃棄可能なすべてのブロックをシステムから強制的に廃棄することができます。

```
HANDLE TheGlobalHandle;
unsigned long BlockSize;
:
// 空きブロックの最大サイズを取得
BlockSize = GlobalCompact(0);
// ロックされていない廃棄可能ブロックをすべて廃棄
GlobalCompact((unsigned long)-1);
```

グローバルメモリブロックの廃棄方法の変更

Windowsはメモリブロックの廃棄が必要となると，LRU (least-recently-used) アルゴリズム（最後に使われてから最も時間が経過しているものを選ぶアルゴリズム）に基づき，どの廃棄可能ブロックを廃棄するかを決定します。***GlobalLRUOldest***関数と***GlobalLRUNewest***関数を使えば，特定のメモリブロックに関して廃棄対象リスト内の位置を指定することができます。***GlobalLRUOldest***関数の呼び出しで指定したブロックが次の廃棄対象ブロックになります。***GlobalLRUNewest***関数は，指定されたブロックを廃棄リストの最後に移動し，廃棄順を最後にします。

メモリ不足に関する警告の受理

Windowsがメモリ圧縮に時間がかかりすぎていると判断した場合，現在動作中のアプリケーションのすべての最上位ウィンドウに対して***WM_COMPACTING***メッセージが送られます。このメッセージは，システムメモリが不足しているため，各アプリケーションに対しできるだけ多くのグローバルメモリを即時解放するよう指示するものです。

プログラミング上の留意点

グローバルメモリブロックを使い終わったらすぐにそのブロックのロックを解除するよういつも心がけてください。ブロックのロックが解除されていないと，そのブロックを移動できないので，以降のメモリ確保に充分な空きメモリを生成できなくなってしまいます。また，プログラムが終了する前にメモリブロックを解放してください。

グローバルヒープのメモリブロックを128バイト未満の単位で確保しないでください。グローバルメモリにはオーバーヘッドがあり，バイトアラインメントの問題もあるからです。また，グローバルメモリブロックを多数確保するということも避けるべきです。その場合は，ブロックを組み合わせて大きなブロックとし，確保するブロック数を減らすようにしてください。

アプリケーション間で通常のグローバルメモリハンドルを受け渡して，共有メモリ領域を作成してはいけません。***GMEM_DDESHARE*** で確保したグローバルメモリだけがアプリケーション間で共有可能です。または，DDEとクリップボードを使ってメモリを共有してください。この点については第17章『**プロセス間通信**』を参照してください。

第17章

プロセス間通信

Windowsでは複数のアプリケーションを同時に実行できるので，実行中にアプリケーション間でデータを共有できると便利です。Windowsではプロセス間通信をサポートするために，クリップボードによる方法，カスタマイズ可能なダイナミックデータエクスチェンジ（DDE：Dynamic Data Exchange）プロトコルを使う方法，Windowsメッセージを新しく定義してこれをアプリケーション間で転送する方法を提供しています。

Windowsのクリップボード

Windowsのクリップボードは，いろいろな形式のデータ貯蔵所として機能します。Windowsアプリケーションでクリップボードにデータを置く場合，定義済みのデータ形式を用いることも，アプリケーション独自のデータ形式を使うこともできます。クリップボードからデータを取り込む場合，アプリケーションはクリップボードに対し，データが現在どのような形式で格納されているか問い合わせることができます。アプリケーションでは，クリップボードにデータを置いたり，クリップボードからデータを取り込む際にいろいろなデータ形式を使えます。

クリップボードのデータ形式

Windowsでクリップボードを用いる場合，定義済みのデータ形式を指定します。この指定はWindowsのクリップボード関数を呼び出すとき，**表17.1**に示す***CF***_定数で行ないます。

表 17.1
クリップボードのデータ形式

データ形式	定義
CF_TEXT	ヌルで終わる文字配列
CF_BITMAP	Windows のビットマップ形式
CF_SYLK	Microsoft の SYLK 形式
CF_TIFF	TIFF 形式
CF_METAFILEPICT	Windows メタファイルピクチャ形式

以上の形式は，既存の Windows アプリケーションで広く採用されている標準的なものです。特殊な独自のデータ形式を使う場合や新しい標準データ形式を確立したい場合は，自分の用いているデータ形式を登録することになります。しかし，クリップボード上に独自の形式でデータを格納した場合は，その形式を扱えるアプリケーションでないとそのデータを取り込むことはできません。

アプリケーションで新しいデータ形式を使う場合，***RegisterClipboardFormat*** 関数を呼び出して新しいデータ形式を（未登録であれば）登録する必要があります。Windows での作業で最初に ***RegisterClipboardFormat*** 関数を呼び出したとき，既存のデータ形式識別子と重複しない新しいデータ形式識別子が返されます。この識別子は上で示した ***CF_*** 定数と同様に用いることができます。そのあと，別のアプリケーションが同じ名前の形式を登録すると，この新しいデータ形式に対しても同じ識別子が返されます。このように，アプリケーションが異なっても，この新しいデータ形式に対する識別子は共通になるわけです。

クリップボードへのデータの格納

クリップボードにデータを置くときは以下の手順で行ないます。

1. クリップボードをオープンする（***OpenClipboard***）。
2. クリップボードを空にする（***EmptyClipboard***）。
3. クリップボードに新しいデータを入れる（***SetClipboardData***）。
4. クリップボードをクローズする（***CloseClipboard***）。

OpenClipboard を呼び出すときは引数としてウィンドウハンドルを渡します。この引数によって，クリップボードを操作するウィンドウはある時点でただ 1 つに限定されます。

クリップボードにデータを置くアプリケーションは，データをいったんグローバルメモリに複写しておき（第16章参照），***SetClipboardData*** を呼び出してそのデータへのハンドルを渡します。ハンドルが渡されると，データの所有権がクリップボードに移行するので，アプリケーションではそのデータを使用できなくなります。したがって，データそのものへのハンドルではなく，データのコピーに対するハンドルを ***SetClipboardData*** 関数に渡すようにしてください。***SetClipboardData*** 関数は，引数として ***CF_TEXT*** などのクリップボードのデータ形式識別子もとります。アプリケーションで ***SetClipboardData*** 関数を繰り返し呼び出し，毎回形式の違うデータを追加することができます。

以下に，クリップボードに ***CF_TEXT*** 形式でデータを置くコードを示します。

```
BOOL TMyWindow::CopyText(LPSTR TextString); {
  HANDLE StringGlobalHandle;
  LPSTR StringGlobalPtr;

  StringGlobalHandle = GlobalAlloc(GMEM_MOVEABLE,
                                   (LONG)lstrlen(TextString) + 1);
  if (StringGlobalHandle)
  {
    StringGlobalPtr = GlobalLock(StringGlobalHandle);
    if (StringGlobalPtr)
    {
      lstrcpy(StringGlobalPtr, TextString);
      GlobalUnlock(StingGlobalHandle);
      if (OpenClipboard(HWindow))
      {
        EmptyClipboard();
        SetClipboardData(CF_TEXT, StringGlobalHandle);
        CloseClipboard();
        return TRUE;
      }
      else GlobalFree(StringGlobalHandle);
    }
    else GlobalFree(StringGlobalHandle);
  }
  return FALSE;
}
```

引数 *TextString* で *CopyText* に渡されるアプリケーションの文字列データが，***lstrcpy*** によりグローバルメモリブロックにコピーされます。したがって，*TextString* に渡されたブロックのハンドルは，まだ使えることになります。呼び出し後，アプリケーションからこのメモリブロックを使うことはできません。

GlobalAlloc を使う場合は，***GMEM_DISCARDABLE*** 以外の属性フラグでグローバルメモリを確保してください。そのグローバルデータが廃棄された場合に，クリップボードではデータを再構築できないからです。

クリップボードからのデータの取り込み

データをクリップボードから取り込むアプリケーションは，以下の 4 つの手順を踏みます。

1. クリップボードをオープンする（***OpenClipboard***）。
2. 現在クリップボードにあるデータの形式を問い合わせる（***IsClipboardFormatAvailable***）。
3. クリップボードからデータを取り込む（***GetClipboardData***）。
4. クリップボードをクローズする（***CloseClipboard***）。

データをクリップボードから取り込むときは，グローバルメモリへのハンドルを使います。データを所有するのはクリップボードであり，データの取り込み後もデータを所有し続けるので，アプリケーションではそのハンドルが示すデータをコピーする必要があります。ブロックをコピーする場合を除き，アプリケーションが直接クリップボードのハンドルを使用したり，操作したりしてはいけません。

以下に，クリップボードから文字列データを取り込む操作を記述した関数の例を示します。

```
int TMyWindow::PasteText(LPSTR TextString, int TextSize) {
  HANDLE StringGlobalHandle;
  DWORD StringGlobalSize;
  LPSTR StringGlobalPtr;
  int CharsPasted;

  CharsPasted = -1;
  if (OpenClipboard(HWindow))
  {
    if (IsClipboardFormatAvailable(CF_TEXT))
    {
      StringGlobalHanddle = GetClipboardData(CF_TEXT);

      if (StringGlobalHandle)
      {
        StringGlobalSize = GlobalSize(StringGlobalHandle);
        StringGlobalPtr = GlobalLock(StringGlobalHandle);

        if (StringGlobalPtr)
        {
          if (TextSize < StringGlobalSize)
            StringGlobalSize = TextSize ;

          _fstrncpy(TextString, StringGlobalPtr, StringGlobalSize);
          TextString[StringGlobalSize] = '¥0';
          GlobalUnlock(StringGlobalHandle);
          CharsPasted = lstrlen(TextString);
        }
      }
      CloseClipboard();
    }
    return CharsPasted;
  }
```

複数のデータ形式を扱えるアプリケーションは，まず最も情報量の多いデータ形式で問い合わせてみるとよいでしょう。たとえば，クリップボードがデータの受け渡しにバイナリデータ形式をサポートしている場合，***CF_TEXT*** 形式があるかないかを問い合わせる前に，まずバイナリデータ形式があるか調べるべきです。バイナリデータではなく文字列形式で数値データを保存すると，数値を文字列に変換したり，その逆を行なう際に丸め誤差が生じる恐れがあるからです。

レンダリングの遅延

多数のデータ形式をサポートしているアプリケーションでは，クリップボードにデータを入れるときに毎回各形式でデータをレンダリング（データ形式を変換すること）しなければならないとすると，かなりの重荷となります。そこで，他のアプリケーションから要求されるまでデータのレンダリングを遅らせることができます。

レンダリングを遅らせるには，***SetClipboardData*** 関数を呼び出す際に，レンダリングを遅らせたいデータ形式のデータハンドルとして 0 を渡します。しかし，レンダリングを遅らせたいアプリケーションは，後でレンダリングするために最後のデータを保存しなくてはなりません。別のアプリケーションが，レンダリングを遅らせるように指定されている形式のデータを要求した場合，Windows はレンダリングを行なうアプリケーションに対し，***WM_RENDERFORMAT*** メッセージを送ります。***WM_RENDERFORMAT*** で渡される ***WORD*** 型引数には，要求されたデータ形式のコードが入っています。アプリケーションはこのメッセージを受け取ると，データを要求された形式でクリップボードに入れます。これは，前に説明したような手順によりますが，クリップボードを空にする操作は行ないません。

レンダリングを行なう方のアプリケーションは，クリップボードの内容が別のアプリケーションによって消去されると，***WM_DESTROYCLIPBOARD*** メッセージを受け取ります。この結果，このアプリケーションはレンダリングのために保存していたデータを廃棄できるようになります。レンダリングを遅らせたアプリケーションが終了する間際に，Windows からそのアプリケーションに ***WM_RENDERALLFORMATS*** メッセージが送られ，レンダリングを遅らせた形式のデータをクリップボードに渡すよう指示されます。

プロセス間でのメッセージ伝達

ウィンドウやアプリケーションが受け取るメッセージの多くは，Windowsや同じアプリケーションの他のウィンドウから送られてくるものです（ユーザ定義メッセージについては第7章を参照してください)。さらに，アプリケーション間でメッセージを送ることもあります。この処理はDDEプロトコルによって形式化されていますが，Windowsは一般的なプロセス間メッセージ伝達機構も提供しています。この機構は，通知メッセージを送ったり，簡単なデータ値を渡す場合に最適です。DDEは，複雑なデータを渡す場合に適しています。

たとえば，電子メールやスケジュール管理，カレンダプログラムなどを含む一連のグループウェアアプリケーションを書いたとします。電子メールアプリケーションには，ネットワークサーバからの電子メール通知メッセージを受け取る特別なルーチンが含まれます。通知メッセージを受け取ると，電子メールプログラムは，スケジュール管理プログラムとカレンダープログラムに電子メールが到着したことを通知します。この場合，***WM_EMAILARRIVED*** という新しいメッセージを定義することができます。この新しいメッセージを送ったり受け取ったりするアプリケーションは，すべてWindows関数 ***RegisterWindowMessage*** を使ってこのメッセージを登録しなければなりません。あるアプリケーションから別のアプリケーションにメッセージを送るには，***SendMessage*** 関数か，***PostMessage*** 関数を使います。オープンしているすべてのウィンドウにメッセージを送るには，ウィンドウハンドルを示す引数として-1を渡します。メッセージの受け取り確認として，受け取り側アプリケーションが別のカスタムメッセージを送り返すようにすることもできます。

しかし，この場合は ***DWORD*** 型引数（*lParam*）でデータへのポインタを渡してはいけません。Windowsがデータを移動させることがあり，そのポインタが常に正しい値を指しているとは限らないからです。また，グローバルメモリハンドルも渡してはいけません。グローバルメモリブロックは，解放されたり，移動したり，廃棄されたりすることがあるからです。メッセージの送り側または受け取り側のアプリケーションがメモリハンドルを解放していない場合にも問題が生じます。原則として，情報を ***WORD*** 型引数や ***DWORD*** 型引数で直接渡すようにしてください。

ダイナミックデータエクスチェンジ

ダイナミックデータエクスチェンジ（DDE：Dynamic Data Exchange）とは，共同で働くアプリケーション同士がデータを共有する際に従うプロトコルのことです。DDEでは，アプリケーションが“共同で”働くということが重要です。DDEプロトコルでは，2つのアプリケーションがともにアプリケーションで定義した共通のトピックに関するデータの要求方法を承知していることが条件になっているからです。DDEは，一群のWindowsメッセージ（***WM_DDE_***で始まるもの）として実現されています。アプリケーションのウィンドウ間でこのメッセージを送ることにより，データを渡したり，別のアプリケーションになんらかの要求を出したりすることができます。なお，DDEを使用するときはプログラムにDDE.Hファイルをインクルードする必要があります。

DDEプロトコルを用いれば，アプリケーション間でグローバルデータを渡す場合につきまとう問題をなくすこともできます。その方法として，グローバルな**アトム**を使って文字列を渡したり，***GMEM_DDESHARE***属性フラグをつけて確保したグローバルメモリを使ってデータを渡したりします。アトムとは，現在実行中のすべてのWindowsアプリケーションがアクセスできるグローバルな文字列定数のことです。

条件

2つのアプリケーション間でデータを交換しようとする場合，一種の話し合いをすることになります。話し合いを持ちかける側のアプリケーションウィンドウをクライアントウィンドウと呼び，この持ちかけに応じる側のアプリケーションウィンドウをサーバと呼びます。

ただし，同じウィンドウを一度に複数の話し合いに従事させないようにしてください。同時に複数の対話を行ないたい場合は，複数のウィンドウを開いて，個々のウィンドウにそれぞれの対話を担当させるようにします。このようにして追加したウィンドウを表示させたくなければ，非表示にすることもできます。

DDEプロトコルでは，交換するデータの仕様は3部構成になっています。まずデータの最初の要素としてアプリケーション名が入り，次にトピックが続き，最後に項目名が入ります。アプリケーションの各インスタンスは，通常特定のアプリケーションのトピックしか“理解”できないので，アプリケーション名を明確にすることは重要です。トピックとしては，Windows上のワードプロセッサでは文書ファイル名を用いるということが考えられます。項目には，要求されたデータ内容が収められます。ワードプロセッサでは，現在編集中の段落や現在使用しているフォント情報などが項目として扱われます。

アプリケーション名とそのトピックによって対話が成立し，項目で1回の対話で送ることができる単位のデータを伝達します。交換されるデータの形式には，クリップボードで用いられる形式であればどの形式でも使用できます（**表17.1**参照）。

対話の成立

クライアントはサーバに働きかけ，対話を開始させます。この対話は，クライアントとサーバのどちらかが対話を終了させるまで続きます。対話は ***WM_DDE_INITIATE*** メッセージで始まり，***WM_DDE_TERMINATE*** メッセージにより終了します。

クライアントは対話を開始するとき，***SendMessage*** 関数で ***WM_DDE_INITIATE*** メッセージを他の全ウィンドウに発行します。この際，ウィンドウハンドルを指定する引数として-1を渡します。DDEプロトコルは非同期であるため，***SendMessage*** 関数は ***WM_DDE_INITIATE*** メッセージを送るとき以外は不要です。どれか特定のウィンドウと対話が始まってしまえば，あとの対話は ***PostMessage*** 関数でメッセージを受け渡します。***PostMessage*** 関数を使ってメッセージを発行した場合は，***SendMessage*** 関数の場合とは異なり，メッセージ送信側のアプリケーションは，他のウィンドウからの応答を待たずに実行を継続できるという特徴があります。ただし，DDEが非同期であることの欠点として，クライアントアプリケーションが，希望する応答が返ってきているかを常に監視（ポーリング）しなくてはならないという点があります。

複数のアプリケーションがサーバーアプリケーションとして ***WM_DDE_INITIATE*** メッセージに応答してくる場合もあります。このような場合，クライアントはその中からサーバとして1つを選択し，他のアプリケーションには ***WM_DDE_TERMINATE*** メッセージを送らなければなりません。このようにして，1つのクライアントに対して応答できるサーバが1つだけになるのです。

WM_DDE_INITIATE によって対話が開始されると，開始側アプリケーションは希望する相手のアプリケーション名と，トピックを指定することができます。アプリケーション名として何も指定しなかった場合は，実行中のどんなアプリケーションとでも対話することができます。トピックをなにも指定しないと，対話を始める際のトピックは何でもよいことになります。

以下の方法を使えば，ウィンドウに対話を開始させることができます。引数としてアプリケーション名とトピックを渡します。何も指定しない場合は引数としてヌルを渡します。

```
void TClientWindow::InitiateConversation(LPSTR AppName,
                                         LPSTR TopicName)
{
  WORD AppGlobalAtom, TopicGlobalAtom;

  if (AppName)
    AppGlobalAtom = GlobalAddAtom(AppName);
  else
    AppGlobalAtom = NULL;
  if (TopicName)
    TopicGlobalAtom = GlobalAddAtom(TopicName);
  else
    TopicGlobalAtom = NULL;
  SendMessage(-1, WM_DDE_INITIATE, HWindow,
             MAKELONG(AppGlobalAtom, TopicGlobalAtom));

  if (AppGlobalAtom)
    GlobalDeleteAtom(AppGlobalAtom);
  if (TopicGlobalAtom)
    GlobalDeleteAtom(TopicGlobalAtom);
}
```

ここでは，***SendMessage*** 関数へは，***DWORD*** 型ポインタではなくグローバルアトムを渡しています。このアトムは，アプリケーション名か内容文字列のどちらかが設定されていれば作成されます。***SendMessage*** 関数を呼び出したあとには，作成したグローバルアトムのために確保したメモリを必ず解放しなくてはいけません。なぜそうしても安全かというと，すべてのアプリケーションウィンドウが ***WM_DDE_INITIATE*** の処理を終えるまで ***SendMessage*** は終了しないからです。グローバルアトムの作成と破壊の規則については，以降で該当する DDE のメッセージごとに説明します。渡したアプリケーション名やトピックに対応できるアプリケーションはすべて，メッセージを確認したという意味の ***WM_DDE_ACK*** メッセージを返します。

ウィンドウオブジェクトは自動メッセージ応答メンバ関数を定義して，***WM_DDE_INITIATE*** メッセージがきたら自動的に応答させることができます。この方法を使えば，指定されたトピックを自動的に調べ，対話したい場合には，別のメンバ関数を呼び出してクライアントウィンドウに確認メッセージを送らせることができます。トピックが何も指定されていない場合，サーバウィンドウはどのトピックで応答するかを決めることができます。どの方法にせよ，サーバアプリケーションは，通常新しいウィンドウを開いて対話を行ないます（DDE による対話ごとにウィンドウが1つずつ対応するということを憶えておいてください)。この新しいウィンドウへのハンドルを確認時に渡すようにします。

次の例は，*TServerWindow* というサーバウィンドウが ***WM_DDE_INITIATE*** メッセージに応答する場合です。

```
void TServerWindow::ACKTopic(HWND ClientHWnd, HWND ConversationHWnd,
                              LPSTR TopicName)
{
  WORD AppGlobalAtom, TopicGlobalAtom;

  AppGlobalAtom = GlobalAddAtom(GetApplication()->Name);
  TopicGlobalAtom = GlobalAddAtom(TopicName);
  if (!SendMessage(ClientHWnd, WM_DDE_ACK, ConversationHWnd,
      MAKELONG(AppGlobalAtom, TopicGlobalAtom)))
  {
    GlobalDeleteAtom(AppGlobalAtom);
    GlobalDeleteAtom(TopicGlobalAtom);
  }
}
```

サーバとしてDDE対話を扱うウィンドウは，ウィンドウオブジェクト*TServerWindow*の***HWindow***のかわりに，*ConversationHWnd*を引数として受け取っています。これは，各トピックごとにウィンドウオブジェクトを新しく作らせるためのものです。このあと，クライアントはどのトピックで対話を継続するかを選択することができます。さらにグローバルアトムが再び生成されますが，このアトムは***SendMessage***関数が失敗したとき以外は削除されません。成功した場合，つまりクライアントが***WM_DDE_ACK***メッセージを受け取ると，クライアント側でグローバルアトムを削除するからです。

この時点で対話が確立したことになります。サーバ側は，***WM_DDE_INITIATE***メッセージを送信したクライアントウィンドウのハンドルを受け取っているので，どのウィンドウがクライアントかを承知しています。クライアント側でも，***WM_DDE_ACK***メッセージを送信したサーバウィンドウのハンドルを受け取っているので，どのウィンドウがサーバかを承知しています。このあとの2つのアプリケーション間の対話は，この2つのハンドルを用いて行なわれます。

対話の終了

サーバウィンドウでもクライアントウィンドウでも，どちらかが***WM_DDE_TERMINATE***メッセージを送れば，対話を終了させることができます。このメッセージは，通常クライアント側から送られます。しかし，アプリケーションがクローズされる場合など，サーバ側から***WM_DDE_TERMINATE***メッセージが送られることもあります。***WM_DDE_TERMINATE***メッセージを受け取ったウィンドウは***WM_DDE_TERMINATE***メッセージを送り返さなくてはなりません。***WM_DDE_TERMINATE***メッセージが処理されたあとに送られてくるDDEメッセージに応答してはいけません。さらに，グローバルアトムやデータもすべて削除しなければなりません。***WM_DDE_TERMINATE***メッセージが返ってこないうちに，送信側ウィンドウをクローズすることはできません。

WM_DDE_TERMINATEメッセージは，致命的エラーでアプリケーションが終了したというような極限状態で送られることもあるので，返信メッセージ***WM_DDE_TERMINATE***が処理されない場合もあります。このような状況での対策としては，***WM_DDE_TERMINATE***を先に送った方のアプリケーションが，***WM_DDE_TERMINATE***が返ってくるまで待つか，あるいは一定の時間が経過するのを待つようにします。***PostMessage***関数を使ってDDE

メッセージを送るときは，常にこういったタイムアウトを設けてください。さらに，***PostMessage*** 関数を呼び出す前に ***IsWindow*** 関数を呼んで，応答先のウィンドウハンドルを示す引数が有効かどうかを調べることもできます。このような予防的措置は，DDE プロトコルの一環というわけではありませんが，クライアントやサーバの壊滅的なダメージから回復する助けとなります。とにかく，用心に越したことはありません。

データ交換に使うメンバ関数

クライアントとサーバでデータを受け渡すには以下の 5 つの方法があります。

1. クライアントが送る ***WM_DDE_REQUEST*** メッセージとサーバが応答に使う ***WM_DDE_DATA*** メッセージで，単一のデータ項目を交換できます。
2. ***WM_DDE_ADVISE*** メッセージと ***WM_DDE_DATA*** メッセージで，クライアントにデータ項目が変化したことを通知することができます。その後，***WM_DDE_REQUEST*** メッセージと ***WM_DDE_DATA*** メッセージを使ってデータを取得します。一般に**ホットリンク**と呼ばれるアプリケーションの機能はこの方法で実現されています。さらに，***WM_DDE_UNADVISE*** メッセージを使って，連続的なデータの更新が終了したことを通知します。
3. 方法 1 と方法 2 を組み合わせることもできます。つまり，***WM_DDE_DATA*** メッセージを連続的に使用して実際にデータを渡します。この場合，***WM_DDE_REQUEST*** は不要となります。
4. ***WM_DDE_POKE*** メッセージで，サーバに対しデータ項目の値を変化させるよう指示することができます。
5. ***WM_DDE_EXECUTE*** メッセージでサーバアプリケーションにコマンドマクロを渡すことができます。サーバ側でそのコマンドマクロを解釈して実行します。

単一データ項目の要求

通常は，クライアントウィンドウがサーバウィンドウにデータを渡すよう要請します。以下に示すコードにより，単一のデータ項目を要求することができます。

```
void TClientWindow::RequestData(WORD DataFormat, LPSTR Item,
                                HWND ServerHWND)
{
  WORD ItemGlobalAtom;

  ItemGlobalAtom = GlobalAddAtom(Item);
  if (!PostMessage(ServerHWND, WM_DDE_REQUEST, HWindow,
      MAKELONG(DataFormat, ItemGlobalAtom)))
    GlobalDeleteAtom(ItemGlobalAtom);
}
```

引数 *DataFormat* は，*CF_TEXT* などのクリップボードで使うことができる形式のうちのどれかです。サーバが要求された形式でデータを渡せる場合は，以下のように ***WM_DDE_DATA*** メッセージを返信します。

```
#include <dde.h>
#include <mem.h>
:
void TServerWindow::ACKData(LPSTR Item, WORD DataFormat, LPSTR Data,
                            WORD DataSize)
{
  DDEDATA far * LPDataRecord;
  WORD  ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                              (LONG)(sizeof(DDEDATA) + DataSize));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEDATA FAR*)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fAckReq = LPDataRecord->fResponse = TRUE;
      LPDataRecord->fRelease = FALSE;
      LPDataRecord->cfFormat = DataFormat;
      _fmemcpy(LPDataRecord->Value, Data, DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom((Item);
      if (!PostMessage(ClientHWND, WM_DDE_DATA, HWindow,
          MAKELONG(DataGlobalHandle, ItemGlobalHandle)))
      {
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      }
```

```
        }
      }
    }
```

この例では注目すべき点がいくつかあります。まず，グローバルメモリが ***GMEM_DDE_SHARE*** 属性で確保されている点があげられます。これによって，サーバとクライアントの双方で，このハンドルが指すデータにアクセスすることができます。次に，グローバルメモリ内の実際のデータの前にデータブロックレコードが書き込まれている点です。これによって，データの形式とデータを受け取るクライアントがどのように応答すればよいかを示すビットフィールドオプションを指定することができます。

2つのオプションを設定できます。1つは *fResponse* で，***WM_DDE_REQUEST*** メッセージに応じて ***WM_DDE_DATA*** メッセージを送ることを示します。もう1つは *fAckReq* で，受け取るクライアントに対し，***WM_DDE_DATA*** メッセージを受け取った確認として，***WM_DDE_ACK*** メッセージを返して欲しいということを指定します。データ形式が ***CF_TEXT*** の場合は，渡されたデータポインタが指すデータの各行が，最後の行も含めて復帰・改行で区切られていなければなりません。

サーバで ***WM_DDE_REQUESTED*** メッセージを処理できない場合は，クライアントに対して処理不可能という通知を送り返さなければなりません。たとえば，サーバが要求された形式でデータ項目を送れない場合などが考えられます。一般的な経験則としては，まず最も凝った形式でデータを要求し，肯定的な通知が返ってくるまでデータ形式を簡略化してゆくという方式をとります。否定通知は以下のようになされます。

```
  ⋮
  ItemGlobalAtom = GlobalAddAtom(Item);

  if (!PostMessage(ClientHWND, WM_DDE_ACK, HWindow /* サーバ側 */,
      MAKELONG(0, ItemGlobalAtom)))
    GlobalDeleteAtom(ItemGlobalAtom);
  ⋮
```

クライアントが ***WM_DDE_DATA*** メッセージを受け取った場合は，サーバから受け取ったデータを処理してよいことを意味します。さらに，データポインタの冒頭にある ***DDEDATA*** レコードもチェックする必要があります。*fAckReq* が設定されていれば，肯定通知をクライアントへ送り返します。

```
:
if (DataRecord->fAckReq)
{
  ItemGlobalAtom = GlobalAddAtom(Item);
  if (!PostMessage(ServerHWND, WM_DDE_ACK, HWindow/* クライアント側 */,
      MAKELONG(0x8000, ItemGlobalAtom)))
  GlobalDeleteAtom(ItemGlobalAtom);
}
:
```

fRelease が設定されている場合，クライアントはメモリハンドルが示すメモリを解放しなくてはなりません。

```
:
GlobalUnlock(ItemGlobalHandle);
if (DataRecord->fRelease)
  GlobalFree(ItemGlobalHandle);
:
```

fAckReq フラグか *fRelease* フラグ，あるいは両フラグを ***WM_DDE_DATA*** メッセージで指定する必要があります。その通知を受け取った場合はサーバが，それ以外ではクライアントがグローバルメモリを解放します。両フラグが立っていないと，サーバはグローバルデータをいつ解放すべきかわかりません。クライアント側でも解放しないことになってしまいます。

これまでの話は，必要以上に複雑すぎる感じを受けたかもしれません。しかし，DDE プロトコルのこの部分は，以下に示す 3 つの重要な事項を実現しようと意図したものなのです。

1. 確保したグローバルメモリオブジェクトがすべて解放されるようにすること。
2. クライアントとサーバのどちらでも利用できるような形式で全グローバルメモリオブジェクトを確保すること。
3. アプリケーションが永遠に他のアプリケーションからの応答を待ち続けるといったデッドロック状態の発生を防ぐこと。

上記の 3 つの目的を各 DDE メッセージでどのように扱っているかを理解すれば，DDE プロトコルの難点はほとんど解消されるはずです。

継続データ転送の維持

クライアント側で，データ項目の変更通知を送るための継続リンク（ホットリンク）をサーバとの間に確立できます。***WM_DDE_ADVISE*** メッセージを送ってこのリンクを確立します。

```
void TClientWindow::Advise(LPSTR Item, WORD DataFormat)
{
  DDEADVISE FAR * LPDataRecord;
  WORD  ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                              (LONG)(sizeof DDEADVISE));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEADVISE FAR *)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fAckReq = LPDataRecord->fDeferUpd = TRUE;
      LPDataRecord->cfFormat = DataFormat;
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom((Item);
      if (!PostMessage(ClientHWND, WM_DDE_ADVISE, HWindow,
          MAKELONG(DataGlobalHandle, ItemGlobalAtom)))
      {
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      }
    }
  }
}
```

サーバが要求された形式でデータを送ることができる場合は，肯定の意味の ***WM_DDE_ACK*** を送り返します。一方，要求に応じられない場合は，否定の意味の ***WM_DDE_ACK*** を送り返します。否定の ***WM_DDE_ACK*** を受け取った場合，クライアントは別のクリップボードデータ形式でデータリンクの確立を試みることができます。

WM_DDE_ADVISE メッセージの *fDeferUpd* フラグが立っている点に着目してください。これは，データの変更があったとき，クライアントはこの変更通知だけを必要とし，データ自体は必要としていないことを示します。この後のサーバからの ***WM_DDE_DATA*** メッセージは，以下の例のようにデータなしで送られてきます。

```
:
ItemGlobalAtom = GlobalAddAtom(Item);
if (!PostMessage(ClientHWND, WM_DDE_DATA, HWindow /* サーバ側 */,
    MAKELONG(0, ItemGlobalAtom)))
  GlobalDeleteAtom(ItemGloalAtom);
:
```

データを必要とする場合，クライアントは ***WM_DDE_REQUEST*** メッセージを送らなければなりません。*fDeferUpd* フラグが立っていなければ，サーバからの後続メッセージ ***WM_DDE_DATA*** では実際にデータも送られてきます。この ***WM_DDE_DATA*** メッセージは前に述べた ***ACKData*** 関数のメッセージに似ていますが，*fRequest* フラグが立っていない点が異なります。

サーバへのデータ値変更要求

```
void TClientWindow::Poke(LPSTR Item, WORD DataFormat, LPSTR Data,
                         WORD DataSize)
{
  DDEPOKE FAR * LPDataRecord;
  WORD ItemGlobalAtom;
  WORD DataGlobalHandle;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)(sizeof DDEPOKE + DataSize));
  if (DataGlobalHandle)
  {
    LPDataRecord = (DDEPOKE FAR *)GlobalLock(DataGlobalHandle);
    if (!LPDataRecord)
      GlobalFree(DataGlobalHandle);
    else
    {
      LPDataRecord->fRelease = FALSE;
      LPDataRecord->cfFormat = DataFormat;
      _fmemcpy(LPDataRecord->Value, Data, DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom = GlobalAddAtom((Item);
```

```
          if (!PostMessage(ServerHWND, WM_DDE_POKE, HWindow,
               MAKELONG(DataGlobalHandle, ItemGlobalHandle)))
          {
            GlobalFree(DataGlobalHandle);
            GlobalDeleteAtom(ItemGlobalAtom);
          }
        }
      }
    }
```

TClientWindow::Poke のコードは，***TServerWindow::ACKData*** に似たようなものです。クライアントからサーバに ***WM_DDE_POKE*** メッセージが送られ，***WM_DDE_DATA*** メッセージは常にサーバからクライアントという方向で送られる点が異なります。サーバは，***WM_DDE_POKE*** メッセージに応じられる場合は肯定通知を，応じられない場合は否定通知を送らなければなりません。サーバは，***WM_DDE_POKE*** メッセージを処理する際に，以下のようにしてデータ項目名を得ることができます。

```
:
char ItemName[128];

ItemGlobalAtom = Msg.LP.Hi;
GlobalGetAtomName(ItemGlobalAtom, ItemName, sizeof ItemName);
:
```

サーバでのマクロコマンドの実行

クライアントは，サーバに実行してほしいヌルで終わる文字列をコマンドマクロとして ***WM_DDE_EXECUTE*** メッセージで送ることができます。サーバがワープロなどのアプリケーションであれば，そのワープロのマクロ言語の形式でコマンド文字列を構成します。他のメッセージと同様，サーバは ***WM_DDE_EXECUTE*** メッセージにどう対処するかについて肯定または否定の通知をしなければなりません。このメッセージを使う例を以下に示します。

```
void TClientWindow::Execute(LPSTR Command) {
  WORD  DataGlobalHandle;
  LPSTR DataGlobalPtr;

  DataGlobalHandle = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                                 (LONG)_fstrlen(Command));
  if (DataGlobalHandle)
  {
    DataGlobalPtr = GlobalLock(DataGlobalHandle);
    if (!DataGlobalPtr)
      GlobalFree(DataGlobalHandle);
    else
    {
      _fstrcpy(DataGlobalPtr, Command);
      GlobalUnlock(DataGlobalHandle);
      if (!PostMessage(ServerHWND, WM_DDE_EXECUTE, HWindow,
          MAKELONG(0, DataGlobalHandle)))
       {
         GlobalFree(DataGlobalHandle);
       }
    }
  }
}
```

送られるコマンド文字列はヌルで終わるものでなければなりません。また，以下のような書式を用いるようにしてください（大カッコで囲んだ部分がコマンド部を，中カッコで囲んだ部分は任意に指定できる要素であることを示します）。

[*macroCall*]{[*macroCall*]}

macroCall は以下のような書式で記述します。

```
macroName(parameter1 {, parameter2 ...})
```

たとえば，ワープロのマクロ言語で，*goto line* (*lineNo*)という命令で指定行へ移動し，その場所の段落を *select current paragraph* という命令で選択できるとします。そこで，50 行目にある段落を選択する場合は，次のようなコマンド文字列を送ることになります。

```
[goto_line(50)][select_current_paragraph]
```

サーバとして DDE をサポートし，さらに強力なマクロ言語を備えたアプリケーションでは，アプリケーション同士の連携プレーの仕方は数えきれません。

システムトピック

DDEをサポートするアプリケーションはすべて「システム」と呼ばれる特別なトピックを扱う必要があります。このシステムトピックによって，サーバとなるアプリケーションの詳細をあまり把握していないクライアントでも扱うことができます。この機構を使えば，自分のアプリケーションがどのようにDDEを実現しているかについて推察できるようになります。以下にシステムトピックがサポートしなくてはならないデータ項目を示します。

1. *SysItems* は，システムトピックの項目リストです。
2. *Topics* は，現在サポートしているトピックの全一覧を示します。アプリケーションによっては，オープンするファイルごとに異なるトピックを扱うという場合もあるので，この一覧は常に変化すると考えられます。
3. *Formats* は，データを渡す際に使用できるクリップボード形式の一覧を示します。

データ項目内の各データ要素間はタブで区切ってください。

第 18 章

GDI について

多くの Windows アプリケーションでは，ウィンドウやダイアログボックス，各種コントロールだけでユーザインターフェースのすべてを構築しています。しかし，グラフィックス描画やイメージ操作などを行なうアプリケーションでは，ウィンドウにグラフィックスを書き込む必要があります。このようなグラフィックスは，線，図形，文字列，ビットマップイメージなどから構成されます。

アプリケーションでグラフィックスを扱えるようにするため，Windows にはグラフィックスデバイスインターフェース（GDI）と呼ばれる一連の関数が用意されています。この GDI は，Windows アプリケーションがグラフィックスを表示したり操作したりする際に利用する，いわばグラフィックスエンジンと考えることもできます。GDI 関数を使えば，表示装置に依存しないグラフィックス描画機能をアプリケーションに持たせることができます。たとえば，ノーマルディスプレイに描画する場合にも，ハイレゾディスプレイに描画する場合にも，さらには Postscript プリンタに出力する場合にもまったく同じ関数を使用できます。これは，デバイスドライバが GDI 関数を実際に使用する出力装置が解釈できるコマンドに翻訳するようになっているので，出力装置に依存しない描画が可能となっているのです。

ディスプレイコンテキスト

今までのDOSベースのグラフィックスプログラムとは異なり，Windowsプログラムは直接画面にピクセルを書き込むようなことはせず，**ディスプレイコンテキスト**と呼ばれる論理的存在へデータを書き込む形をとります。ディスプレイコンテキストとは，ペン，ブラシ，フォント，バックグラウンド色，文字色，現在位置などの属性を備えた仮想的な画面のことです。

GDI関数を呼び出してディスプレイコンテキスト上に描画すると，ディスプレイコンテキストに対応するデバイスドライバが描画動作をコマンドに変換します。このコマンドによって，表示装置上にできる限り正確に描画動作を再現します。この際，表示能力には影響されません。したがって，表示装置としては低解像度のモノクロームスクリーンでも，24ビットのカラースクリーンでもかまいません。

ウィンドウを額縁つきの油絵とすると，ディスプレイコンテキストはキャンバスに相当するものです。普通は，額縁をつけたまま絵を描かずに，まずキャンバスに絵を描いてから額縁をはめます。ウィンドウのディスプレイコンテキストにグラフィックスを描く場合もまったく同じことです。ディスプレイコンテキストは，ペン，ブラシ，フォントなどの**描画ツール**に現在どれを使っているか管理します。

ディスプレイコンテキストは，ウィンドウ要素と同様，Windowsが管理する要素の1つです。ただし，ディスプレイコンテキストに対応するObjectWindowsオブジェクトはありません。

ディスプレイコンテキストの管理

ディスプレイコンテキストにグラフィックスを描く場合，プログラムではまず希望するウィンドウのディスプレイコンテキストを取得しておく必要があります。しかし，ディスプレイコンテキストは膨大なメモリ量を必要とするので，Windowsの各セッションで同時に使えるディスプレイコンテキストは5つまでです。すなわち，各ウィンドウごとにディスプレイコンテキストを保持しておくことはできません。必要なときだけディスプレイコンテキストを取得し，不要になったら即座に解放しなくてはなりません。

ディスプレイコンテキストの取得，使用，解放の方法については，この章の節「**ウィンドウでのグラフィックス表示**」で詳しく解説します。

ディスプレイコンテキストの構成要素

通常，ディスプレイコンテキストの属性を変更する必要はほとんどありませんが，ディスプレイコンテキストの内容は知っておいてください。この節では，ディスプレイコンテキストに含まれるビットマップ，色，マッピング，クリッピング，描画ツールなどの特徴を簡単に説明します。その中には，この章の別のところでさらに詳しく説明しているものもあります。

ビットマップ

ディスプレイコンテキストで実際に描画される面を**ビットマップ**と呼びます。ビットマップには装置のメモリ構成が反映されます。したがって，ビットマップはどんな種類の装置がメモリ上に配置されているかに依存します。この点は問題だといえます。つまり，ある装置を使って保存したビットマップと別の装置で保存したものとでは互換性がない場合も考えられます。そこで，GDI では**機種非依存型ビットマップ**などの技術を用いてこの問題に対処しています。ビットマップを生成するGDI関数には，***CreateCompatibleDC***, ***CreateCompatibleBitmap***, ***CreateDIBitmap*** などがあります。ビットマップを操作する GDI 関数としては，***BitBlt***，***StretchBlt***，***StretchDIBits***，***SetDIBitsToDevice*** があります。

色

装置が描画に使用する色は**カラーパレット**に保持されています。使いたい色がカラーパレットの中にない場合は，その色を追加することもできます。また，デバイスドライバにカラーパレット内の色をディザリングして，希望する色に近似させることもできます。カラーパレットについては，この章の 330 ページの節「**パレットの使い方**」でさらに詳しく説明しています。

マッピングモード

表示装置としてなにが使われるかがわからないと，描画用の単位を設定しにくいといえます。大部分のアプリケーションはこの点について配慮しておらず，デフォルトの単位（1ピクセル）で充分だとしているようです。しかし，アプリケーションによっては，イメージの大きさを正確に再現した表示が必要となることもあります。このようなアプリケーションのために，GDIではいろいろな**マッピングモード**を用意しています。装置に依存しないモードもあります。

各マッピングモードで単位と座標系が設定されます。デフォルトのマッピングモードは，ディスプレイコンテキストの左上隅を原点として，右方向が正のX軸方向，下方向が正のY軸方向となるように設定されます。ディスプレイコンテキストにはそれぞれマッピングモード属性が定義されており，指定した座標の解釈の仕方をこの属性によって判断します。

ときには，描画の際に用いた論理座標と，ビットマップ上の物理座標との間の変換が必要になります。つまり，現在使用しているマッピングモードによっては，論理座標からビットマップ上の物理座標に変換しなければならないこともあります。アプリケーションのほとんどが画面の原点を左上隅としていますが，ウィンドウの原点はウィンドウのクライアント領域の左上隅です。ウィンドウによっては，クライアント面をスクロールして原点がクライアント領域にない場合もあります。GDI関数には特定の座標系でしか働かないものもあるので，座標変換が必要となります。そのため，GDIにはこういった座標変換を扱う関数として ***ScreenToClient***，***ClientToScreen***，***DPtoLP***，***LPtoDP*** があります。

クリッピング領域

意図した領域の外には描画しないようにするため，ディスプレイコンテキストにはそれぞれ**クリッピング領域**属性が備わっています。クリッピング領域を複雑な多角形や楕円形にして，ディスプレイコンテキストの仮想面上の描画内容をこの中に表示させることもできます。アプリケーションのほとんどは，デフォルトのクリッピング領域であるウィンドウのクライアント領域で充分なはずです。クリッピング領域を変更する必要があるのは，特殊な視覚効果を期待するアプリケーションぐらいなものでしょう。

描画ツール

実際に描画する場合，ディスプレイコンテキストは**ペン**，**ブラシ**，**フォント**の3つの道具（ツール）でほとんどの作業を行ないます。ペンを使って直線，曲線，複数の線分からできた折れ線などを描きます。ペンの属性には，色，幅，スタイル（実線，点線など）があります。長方形や面取りした長方形，楕円や多角形といった図形を一様に塗りつぶすにはブラシを使います。ベタ塗り図形用のGDI関数は，ペンで縁どりを描き，ブラシで中を塗りつぶします。ブラシは4種類あります。ベタ塗り用，ハッチパターン用，ビットマップパターン用，装置独立ビットマップパターン用の4つです。フォントツールは，ディスプレイコンテキストに文字を書く際に用います。フォントツールでは，フォントの高さ，幅，間隔，文字種，書体名を指定できます。

3つのツールはすべてディスプレイコンテキストの**バックグラウンド色**属性で隙間を塗りつぶします。描画ツールについては，次の節「**描画ツール**」で詳しく説明しています。

描画ツール

ディスプレイコンテキストは，画面上のグラフィックス表示を管理します。様々な方法でグラフィックスを表示するために，グラフィックスを描く際に用いる描画ツールを変更することができます。描画ツールの属性は，***LineTo***，***Rectangle***，***TextOut*** といったGDI関数で描かれるグラフィックスの外観を左右します。ペンは，線の描かれ方，ブラシは図形の内側の外観，フォントは描画文字列の外観を規定します。

描画ツールに属性を付与するには，Windowsアプリケーションでディスプレイコンテキストに対する論理ツールかストックツールを選択します。論理ツールとは，プログラムで特定のレコード（***TLOGPEN***，***TLOGBRUSH***，***TLOGFONT***）のフィールドを埋めることによって生成されます。一方，ストックツールはWindowsの定義済みツールであり，黒い実線用のペン，白のブラシ，システムフォントなど，最も汎用的な属性が反映されています。

ストックツール

ストックツールのハンドルは GDI 関数 ***GetStockObject*** で次のように取り込むことができます。

```
HBRUSH TheBrush;
 :
  TheBrush = GetStockObject(LTGRAY_BRUSH);
 :
```

上のコードで ***LTGRAY_BRUSH*** は整数型定数であり，WINDOWS.H に定義されています。次の表はストックツール定数の一覧です。

表 18.1　ストック描画ツール

ブラシ	ペン	フォント	パレット
WHITE_BRUSH	***WHITE_PEN***	***OEM_FIXED_FONT***	***DEFAULT_PALLETE***
LTGRAY_BRUSH	***BLACK_PEN***	***ANSI_FIXED_FONT***	
GRAY_BRUSH	***NULL_PEN***	***ANSI_VAR_FONT***	
DKGRAY_BRUSH		***SYSTEM_FONT***	
BLACK_BRUSH		***DEVICE_DEFAULT_FONT***	
NULL_BRUSH		***SYSTEM_FIXED_FONT***	
HOLLOW_BRUSH			

☞　論理ツールとは異なり，ストックツールを使用後に削除してはいけません。

論理ツール

論理ツールの構造体 ***LOGPEN***，***LOGBRUSH***，***LOGFONT*** には各ツール属性を保持するフィールドがあります。たとえば，***LOGPEN.lopnColor*** には，ペンのカラー属性が収められます。ツールの種類ごとに固有の属性が各構造体で定義されます。

論理ツールを作成するには，ツール属性を定義した論理ツール構造体を GDI 作成関数に渡します。

論理ペン

GDI 関数の ***CreatePen*** や ***CreatePenIndirect*** によって論理ペンを定義します。たとえば以下のように行ないます。

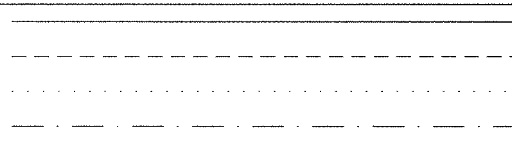
```
ThePen = CreatePen(PS_DOT, 3, RGB(0, 0, 210));
ThePen = CreatePenIndirect(&ALogPen);
```

LOGPEN 構造体は以下のように定義されています。

```
typedef struct tagLOGPEN {
  WORD   lopnStyle;
  POINT  lopnWidth;
  DWORD  lopnColor;
} LOGPEN;
```

lopnStyle はスタイルを示すフィールドです。以下のラインスタイルのいずれかを示す定数を保持します。

図 18.1
ペンツール用のラインスタイル

定数	描画
PS_SOLID	
PS_DASH	
PS_DOT	
PS_DASHDOT	
PS_DASHDOTDOT	
PS_NULL	

lopnWidth は幅を示すフィールドです。このフィールドの X 座標は表示装置の座標系での線幅を示す整数です。この値が 0 の場合，ハイレゾディスプレイ上で描かれる線の幅は 1 ピクセルとなります。Y 座標の値は無視されます。

lopnColor は色を示すフィールドです。赤，緑，青の 3 つの基本色の輝度を示す ***DWORD*** が保持されます。この 3 つの色を組み合わせて，希望する色を生成します。この ***lopnColor*** の値は，0x00*bbggrr* の形で指定しなくてはなりません。*bb* で青の値を，*gg* で緑の値を，*rr* で赤の値を指定します。各色の輝度範囲は 10 進数で 0 から 255，16 進数では 0 から FF までの間です。下の表に，色指定の例を示します。

表 18.2
RGB カラー値の例

値	色
0x00000000	黒
0x00FFFFFF	白
0x000000FF	赤
0x0000FF00	緑
0x00FF0000	青
0x00808080	灰

lopnColor 以外に，***RGB*** マクロを使って色を生成することもできます。たとえば，***RGB***(0, 0, 0) は黒を，***RGB***(255, 0, 0) は赤を指定するといった使い方ができます。

論理ブラシ

GDI関数 ***CreateHatchBrush***, ***CreatePatternBrush***, ***CreateDIBPatternBrush***, ***CreateBrushIndirect*** によって論理ペンを定義します。たとえば以下のように行ないます。

```
TheBrush = CreateHatchBrush(HS_VERTICAL, RGB(0, 255. 0));
TheBrush = CreateBrushIndirect(&ALogBrush);
```

LOGBRUSH 構造体は以下のように定義されています。

```
typedef struct tagLOGBRUSH {
  WORD   lbStyle;
  DWORD  lbColor;
  int    lbHatch;
} LOGBRUSH;
```

lbStyle フィールドは，以下に示すようなブラシのスタイルを示す ***WORD*** 型の定数値をとります。

- ***BS_DIBPATTERN*** を指定すると，装置独立ビットマップで定義されたパターンブラシになります。
- ***BS_HATCHED*** を指定すると，定義済みのハッチパターンの1つになります（***lbHatch*** を参照のこと）。
- ***BS_HOLLOW*** を指定すると何も描かないブラシとなります。
- ***BS_PATTERN*** を指定すると，メモリビットマップの左上隅8×8を使用します。

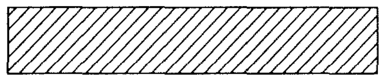
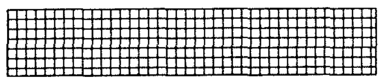
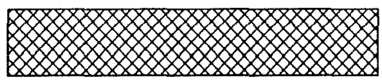
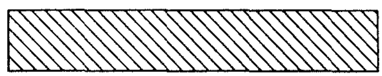
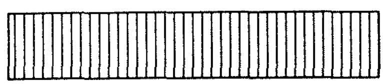
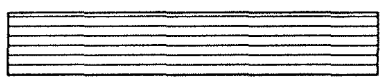
■ ***BS_SOLID*** を指定するとベタ塗りのブラシとなります。

lbColor フィールドは色を示す値を保持します。この色の値は ***LOGPEN*** レコードで要求されたものと同様です。このフィールドは，ブラシのスタイルとして ***BS_HOLLOW*** や ***BS_PATTERN*** が指定された場合には無視されます。

ビットマップリソースをブラシとして使う方法については350ページを参照してください。

lbHatch フィールドは，ブラシのスタイルとして ***BS_HATCHED*** が指定された場合のハッチパターンを示す整数型定数を保持します。ブラシのスタイルとして ***BS_DIBPATTERN*** が指定された場合，***lbHatch*** はビットマップへのハンドルを保持します。

図 18.2
ブラシツールのハッチパターンの種類

定数	描画
HS_BDIAGONAL	
HS_CROSS	
HS_DIAGCROSS	
HS_FDIAGONAL	
HS_HORIZONTAL	
HS_VERTICAL	

論理フォント

GDI 関数の ***CreateFont*** または ***CreateFontIndirect*** によって論理フォントを定義します。

LOGFONT 構造体は以下のように定義されています。

```
typedef struct tagLOGFONT {
  int     lfHeight;
  int     lfWidth;
  int     lfEscapement;
  int     lfOrientation;
  int     lfWeight;
  BYTE    lfItalic;
  BYTE    lfUnderline;
  BYTE    lfStrikeOut;
  BYTE    lfCharSet;
  BYTE    lfOutPrecision;
  BYTE    lfClipPrecision;
  BYTE    lfQuality;
  BYTE    lfPitchAndFamily;
  BYTE    lfFaceName[LF_FACESIZE]
} LOGFONT;
```

フォントツールを作成する際に ***LOGFONT*** を用いる場合は，希望するフォントの属性を指定します。ただし，自分のプログラムがその情報を得たり，その情報から画面用のフォントを生成するわけではありません。要求したフォントがWindows ですでに定義されている画面フォントに対応づけられます。

lfHeight フィールドでフォントの高さを指定します。この値が 0 の場合はデフォルトの大きさになります。正の値の場合は要求したセルの高さを論理単位で表わします。負の値の場合は正の値に直され，文字の高さを論理単位で表わします。

lfWidth フィールドでは，フォント幅を物理単位で指定します。この値が 0 の場合，アスペクト比は変更されません。

文字列の回転を指定する場合，反時計方向に 0.1 度単位で ***lfEscapement*** フィールドに設定します。***lfOrientation*** フィールドは，同様な回転を各文字に対して指定します。

lfWeight フィールドでウェイト（文字の線の太さ）を指定します。この場合，***FW_LIGHT***，***FW_NORMAL***，***FW_BOLD***，***FW_DONTCARE*** などのフォントウェイト定数を使うか，または 0 から 1000 までの値を使います。

イタリック，下線，ストライクアウトの 3 つの属性を要求するときは，それぞれ ***lfItalic***，***lfUnderline***，***lfStrikeOut*** を 0 以外にします。

lfCharset フィールドにより，特定の文字セットを指定することができます。文字セットとして，***ANSI_CHARSET***，***OEM_CHARSET***，***SYMBOL_CHARSET***，***SHIFTJIS_CHARSET***，***HANGEUL_CHARSET*** を指定できます。***OEM_CHARSET*** はシステムに依存します。

lfOutPrecision フィールドでは，Windowsが提供するフォントの大きさや位置がプログラムで指定した値にどの程度正確に一致しなくてはならないかを指定します。デフォルトでは ***OUT_DEFAULT_PRECIS*** です。***lfClipPrecision*** フィールドは，クリップ領域の外にはみ出した文字をどう処理するかを指定します。デフォルトは ***CLIP_DEFAULT_PRECIS*** です。

lfQuality フィールドでは，Windowsが提供するフォントがプログラムで指定した属性値にどの程度一致しなくてはならないかを指定します。指定する値として，***DEFAULT_QUALITY***, ***DRAFT_QUALITY***, ***PROOF_QUALITY*** の3つを使用することができます。***PROOF_QUALITY*** を指定したとき，太字，イタリック，下線つき，ストライクアウトつきフォントが利用できなければ，他の文字から合成されます。

lfPitchAndFamily フィールドでピッチとフォントの文字種を指定します。指定する際には，ピッチを示す定数と文字種を示す定数の **OR**（|）ビット演算値を使うことができます。

表 18.3
フォントのピッチと文字種を示す定数

ピッチ定数	文字種定数
DEFAULT_PITCH	***FF_MODERN***
FIXED_PITCH	***FF_ROMAN***
VARIABLE_PITCH	***FF_SCRIPT***
	FF_SWISS
	FF_DECORATIVE
	FF_DONTCARE

最後の ***lfFaceName*** フィールドには，要求する書体を指定するための文字列を代入します。ゼロを指定した場合は，***LOGFONT*** の他のフィールドの値をもとにして設定された書体が用いられます。

以下に ***LOGFONT*** レコードで定義したフォントの例をソースコードとともにいくつか示します。

Sample Text

```
void MyWindow::MakeFont()
{
  LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 30;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_BOLD;
  MyLogFont.lfItalic         = 0;
  MyLogFont.lfUnderline      = 0;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = ANSI_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = DEFAULT_QUALITY;
  MyLogFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
  strcpy(MyLogFont.lfFaceName, "Helv");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

Sample Text

```
void MyWindow::MakeFont()
{
 LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 10;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_NORMAL;
  MyLogFont.lfItalic         = TRUE;
  MyLogFont.lfUnderline      = TRUE;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = ANSI_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = DEFAULT_QUALITY;
  MyLogFont.lfPitchAndFamily = FIXED_PITCH | FF_DONTCARE;
  strcpy(MyLogFont.lfFaceName, "Courier");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

Σαμπλε Τεξτ

```
void MyWindow::MakeFont()
{
  LOGFONT MyLogFont;

  MyLogFont.lfHeight         = 30;
  MyLogFont.lfWidth          = 0;
  MyLogFont.lfEscapement     = 0;
  MyLogFont.lfOrientation    = 0;
  MyLogFont.lfWeight         = FW_NORMAL;
  MyLogFont.lfItalic         = 0;
  MyLogFont.lfUnderline      = 0;
  MyLogFont.lfStrikeOut      = 0;
  MyLogFont.lfCharSet        = SYMBOL_CHARSET;
  MyLogFont.lfOutPrecision   = OUT_DEFAULT_PRECIS;
  MyLogFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  MyLogFont.lfQuality        = PROOF_QUALITY;
  MyLogFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
  strcpy(MyLogFont.lfFaceName, "Tms Rmn");

  TheFont = CreateFontIndirect(&MyLogFont);
}
```

ウィンドウでのグラフィックス表示

ペイントとは，ウィンドウの内容を表示する処理のことをいいます。ウィンドウが初めて表示される場合と更新が必要な場合は，Windows アプリケーションはそれ自身のペイントに対する責任を持たなければなりません。アイコン化されていたウィンドウが復元されるとき，別のウィンドウに隠されていた部分が見えるようになったとき，あるいはサイズが変更されたときに更新が必要になります。Windows にはウィンドウ内容を自動的にペイントし直すという機能はなく，ウィンドウのペイントが必要になるとウィンドウにその旨を通知します。この節では，ウィンドウでの描画方法，ペイントのメカニズム，ディスプレイコンテキストの使用法を説明します。

この節では，**ペイント**と**描画**とは，ともにウィンドウにグラフィックスを表示する意味で使います。ただし，**ペイント**はウィンドウが最初に出現するとき，または更新が必要なときのグラフィックスの自動表示を指します。これ以外の場合は**描画**といい，直接プログラムによって特定のグラフィックスを生成したり，表示したりする場合を指します。**グラフィックス**とは，文字要素とグラフィック要素（ビットマップや長方形など）の両方を意味します。

ウィンドウでの描画

文字列やグラフィックスをウィンドウオブジェクトに描画するには，まずディスプレイコンテキストを**取得**する必要があります。また描画後には，ディスプレイコンテキストを**解放**しなくてはなりません（1つのWindowsセッションで，同時に5つのディスプレイコンテキスト要素しか利用できません）。ハンドルを取得してから解放するまでの間は，GDI関数を呼び出す際にディスプレイコンテキストハンドルを引数として用いることができます。

ディスプレイコンテキストの操作

通常，ディスプレイコンテキストへのハンドルを保持するウィンドウデータメンバを定義します。これは，***HWindow***がウィンドウハンドルを保持するのとよく似ています。以下に例を示します。

```
class TMyWindow : public TWindow
{
  HDC TheDC;
 ⋮
};
```

ウィンドウ用のディスプレイコンテキストを取得するには，以下のようにWindows関数***GetDC***を呼び出します。

```
TheDC = GetDC(HWindow);
```

ディスプレイコンテキストを取得すると，***Rectangle***などのGDI関数にディスプレイコンテキストハンドルを渡して，以下のようにディスプレイコンテキスト上で描画操作を行なうことができます。

```
Rectangle (TheDC, 10, 10, 100, 1000);
```

ディスプレイコンテキストでの作業が終わりしだい，***ReleaseDC***関数を呼び出してディスプレイコンテキストを解放してください。

```
ReleaseDC(HWindow, TheDC);
```

RealseDC 関数を呼ばずに，***GetDC*** 関数を続けて2度呼び出してはいけません。このようなことをすると，プログラムの実行中に利用可能な5つのディスプレイコンテキストがすべてなくなり，システムエラーが発生します。

Windows グラフィック関数の呼び出し

通常は，ウィンドウのメンバ関数からGDI関数を呼び出します。たとえば，***TextOut*** はディスプレイコンテキストの特定の位置に文字列を描きます。

```
void TMyWindow::DrawText
{  TheDC = GetDC(HWindow);
   TextOut(TheDC, 50, 50, "Sample Text", 11);
   ReleaseDC(HWindow, TheDC);
}
```

ウィンドウのペイント

ウィンドウのペイントが必要になるのは，そのウィンドウが**無効化**されているときです。すなわち，現在表示されている内容がもはや有効ではなく，更新する必要があるときです。このような状況は，ウィンドウが初めて表示される場合や，アイコン化されていたウィンドウを復元させる場合，あるいは別のウィンドウの下に隠れていた部分が出てきた場合などに発生します。いずれの場合でも必ずWindowsが，対象となるアプリケーションに対し ***WM_PAINT*** メッセージを送ります。

TWindow には，***WM_PAINT*** メッセージへの応答メンバ関数 ***TWindow::WMPaint*** が定義されており，主に以下のことを行ないます。

- ディスプレイコンテキストを取得する。
- 仮想メンバ関数 ***Paint*** を呼び出し，ディスプレイコンテキストを引数として渡す。
- ディスプレイコンテキストを解放する。

TWindow のメンバ関数 ***Paint*** は何もペイントしません。***TWindow*** オブジェクトが，デフォルトで何もグラフィックを出力しないからです。ウィンドウにグラフィックスや文字列を描画するには，メンバ関数 ***Paint*** を定義し直して描画用の GDI 関数を呼び出すようにしてください。

ディスプレイコンテキストに対しては，他の色のペンや他のパターンのブラシなど，新しい描画ツールを選択することだけが許されます。これらのツールをメンバ関数 ***Paint*** でペイント用のディスプレイコンテキストに再設定しなくてはなりません。

グラフィックスの描画方法

メンバ関数 ***Paint*** は，最初にウィンドウが出現する場合を含めて，いつでもウィンドウの現時点の内容を描画できるようになっていなくてはなりません。このためメンバ関数 ***Paint*** は，ウィンドウで“常に表示される”グラフィックスをすべて描画できるようにしなくてはなりません。さらに，最初に表示されたウィンドウに後で追加されたグラフィックスも作成し直せることが必要です。このような“状況に応じて変化する”グラフィックスを描けるようにするため，メンバ関数 ***Paint*** は，グラフィックスを生成した際に使った命令やデータを利用できるようにする必要があります。

それには 2 つの方法があります。1 つの方法は，グラフィックスを描くコードを抜き出していくつかのメンバ関数に分割しておき，グラフィックスを生成するときに必要に応じて，メンバ関数 ***Paint*** から呼び出すというものです。もう 1 つは，第 3 章のステップ 5 に例を示したように，ウィンドウに描くグラフィックスに関するデータをウィンドウオブジェクトのメンバに保存するという方法です。データとしては，座標，式，ビットマップなどが該当します。メンバ関数 ***Paint*** では，このデータをグラフィックスに変換する際に必要なグラフィックス用ルーチンを呼び出すようにします。

こういった方法と，オブジェクトのデータと関数を保持できるという性質を利用することにより，生き生きとしたグラフィックスアプリケーションを開発することができます。

描画ツールの使い方

ディスプレイコンテキストは，ウィンドウの描画ばかりでなく，ペン，ブラシ，フォント，パレットといった描画ツールも提供します。これらのツールを用いて文字やグラフィックスを描くことができます。ディスプレイコンテキストに線を描くと，現在選択されているペンの属性で描かれます。つまり，線の属性がそのペンの色，種類（実線か破線かなど），太さになります。領域を描画する場合も，ブラシの塗りつぶしパターンや色など，現在選択されているブラシの属性で塗りつぶされます。ディスプレイコンテキストに文字を描くと，現在選択されているフォントツールのフォント（モダーン，ローマン，スイスなど），大きさ，スタイル（イタリック，太字など）で描かれます。パレットには，現時点で利用可能な色が保持されています。

ディスプレイコンテキストは，各種の描画ツールを１つずつ保持しています。新しく取得したディスプレイコンテキストは，デフォルトのツールとして，黒い細線を描くペン，白のベタ塗り用ブラシ，システムフォント，デフォルトのパレットを備えています。このデフォルトのツールでよければ変更する必要はありません。

このデフォルトのツールを変更したい場合は，新しいツール要素へのハンドルを取得して，これをディスプレイコンテキストに組み込まなければなりません。たとえば，新しいペンを選択すると，もとの古いペンは自動的に削除されます。それまで使っていたツールを保管しておき，新しいツールを使い終わったら，再びもとのツールを選択し直すようにすることをお勧めします。以下にその方法を例で示します。

```
{
  HPEN NewPen, OldPen;
  HDC TheDC;
  // 太さが10のペンを作成する
  NewPen = CreatePen(PS_SOLID, 10, RGB(0,0,0));
  TheDC = GetDC(AWindow->HWindow);
  OldPen = SelectObject(TheDC, NewPen);

  // 描画する
  selectObjet(TheDC, OldPen);
  ReleaseDC(AWindow->HWindow, TheDC);
  DeleteObject(NewPen);
}
```

上の例で示したように，描画ツールを新しく生成したのち，削除する必要があります。ディスプレイコンテキストと同様，描画ツールも Windows のメモリに格納される要素です。描画ツールを削除しないとメモリが不足し，やがては重大なエラーを引き起こすことになります。また，ディスプレイコンテキストと同様，描画ツールへのハンドルもそれぞれ ***HPEN*** 型，***HBRUSH*** 型，***HFONT*** 型，***HPALETTE*** 型の各変数に格納してください。

Windows 関数 ***DeleteObject*** は，Windows メモリから描画ツールを削除します。**現在選択され，ディスプレイコンテキストに組み込まれている描画ツールを削除してはいけません！**

ディスプレイコンテキストは，一度にそれぞれ1つずつしか描画ツールを選択，保持することはできませんが，利用可能な描画ツールを多数保持することは可能です。ただし，必ずアプリケーションが終了する前に，全描画ツールを削除しなければなりません。その方法として，第3章の例で示したように，現在使用中のペンツールへのハンドルを格納する *ThePen* というウィンドウデータメンバを定義することが考えられます。ユーザが別のスタイルのペンを新しく選択すると，新しいペンツールが生成され，もとのペンツールは削除されます。最終的には，メインウィンドウのデストラクタにより最後のペンが削除されます。新しく取得したディスプレイコンテキストによって提供されたデフォルトのツールを削除してはいけません。

新しい描画ツールへのハンドルを取得する方法は2つあります。簡単なのは，**ストック**ツールと呼ばれる既存のツールを利用する方法です。**表 18.1** はストックツールの一覧です。

ストックツールをディスプレイコンテキストオブジェクトに設定するには，***SetStockPen***，***SetStockBrush***，***SetStockFont***，***SetStockPalette*** の各メンバ関数を使用します。たとえば以下のように行ないます。

```
ThePen = GetStockObject(BLACK_PEN);
```

カスタムツールの場合とは異なり，ストックツールを Windows メモリから削除してはいけません。

目的の属性を持ったストックツールがない場合も考えられます。たとえば，ストックペンには細線を引くものしかありませんが，太線を引きたい場合もありえます。そのような場合にカスタム描画ツールを作成する方法が2つあります。1つは，Windows 関数の ***CreatePen***，***CreateFont***，***CreateSolidBrush***，***CreateDIBPatternBrush*** を呼び出す方法です。これらの関数は，目的のツールを記述した引数をとり，***SelectObject*** の引数として使用するツールのハンド

ルを返します。

カスタム描画ツールを生成するもう1つの方法では，**論理ツール**と呼ばれるツールの属性を記述します。論理ツールは，Windowsのデータ構造体***LOGPEN***，***LOGBRUSH***，***LOGFONT***，***LOGPALETTE***によって記述されます。たとえば***LOGPEN***構造体には，太さ，色，スタイルに関するフィールドがあります。論理ツールのデータ構造体を設定し，このデータ構造体を引数にして***CreatePenIndirect***，***CreateBrushIndirect***，***CreateFontIndirect***，***CreatePalette***を呼び出します。これらの関数は，***SelectObject***の引数として使用するツールのハンドルを返します。以下の例では，Windowsのディスプレイコンテキストのペンの色を青色に設定しています。

```
void SampleWindow::ChangePenToBlue()
{
  LOGPEN ALogPen;
  HPEN ThePen;

  AlogPen.lopnColor = RGB(0, 0, 255);
  ALogPen.lopnStyle = PS_SOLID;
  AlogPen.LopnWidth.X = 0;
  AlogPen.lopnWidth.Y = 0;
  ThePen = CreatePenIndirect(&ALogPen);
  SelectObject(TheDC, ThePen);
}
```

GDI描画関数

この節では，ウィンドウに描画する際に使用できるGDI関数の呼び出しについて説明します。

文字列の描画

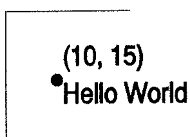
文字列を描画するGDI関数は，現在ディスプレイコンテキストで設定されているフォントを用いて処理を行ないます。***TextOut***関数は，指定した位置に文字列を描き，さらに現在設定されている文字列書式フラグの値に従って文字列の位置を揃えます。デフォルトは左詰めとなります。現在の文字揃え位置は***GetTextAlign***関数で知ることができます。一方，文字揃え位置を設定する場合は***SetTextAlign***関数を使います。

TextOut 関数は，文字列描画を行なう場合に最も多く使われる関数です。文字列書式フラグのデフォルト設定を使うと，以下に示すメンバ関数 ***Paint*** は，ウィンドウのクライアント領域の座標位置（10，15）に左上隅がくるように左詰めで文字列を描きます。

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  char MyTextString[20];

  strcpy(MyTextString, "Hello, World");
  TextOut(PaintDC, 10, 15, MyTextString, strlen(MyTextString));
}
```

図 18.3
TextOut 関数の結果



線の描画

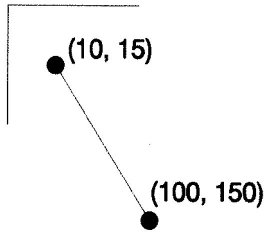
線を描画する GDI 関数は，現在ディスプレイコンテキストで設定されているペンを用いて線を引きます。ほとんどの場合，***MoveTo*** 関数か ***LineTo*** 関数で線が引かれます。この 2 つの関数は，ディスプレイコンテキストの**現在位置**属性に影響します。鉛筆で紙に線を引く場合を考えてみてください。現在位置とは，鉛筆が紙に触れる位置のことをいいます。

MoveTo と LineTo

MoveTo は，現在位置を指定の座標位置へ移動させます。一方，***LineTo*** 関数は，ペンで現在位置から指定の座標位置まで線を引きます（指定座標位置自体は含みません）。この指定座標位置が次の現在位置になります。以下に示すメンバ関数 ***Paint*** は，（100，150）から（10，15）へ線を引きます。

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  MoveTo(PaintDC, 100, 150);
  LineTo(PaintDC, 10, 15);
}
```

図 18.4
LineTo 関数の結果



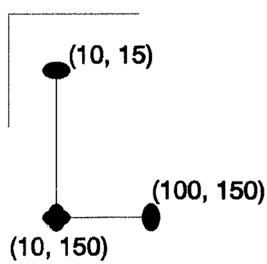
連続線

Polyline 関数は，点をつなぐようにして連続した線を描きます。この関数では，***MoveTo*** 関数を使ってから ***LineTo*** 関数を数回使った場合と同じ描画結果が得られます。ただし，***Polyline*** 関数を使うと描画がはるかに高速なうえ，しかも現在位置が変化しないという特色があります。以下に示すメンバ関数 ***Paint*** は，右開きの L 字線を引きます。

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  POINT Points[3];

  Points[0].x = 10;
  Points[0].y = 15;
  Points[1].x = 10;
  Points[1].y = 150;
  Points[2].x = 100;
  Points[2].y = 150;
  Polyline(PaintDC, Points, 3);
}
```

図 18.5
Polyline 関数の結果



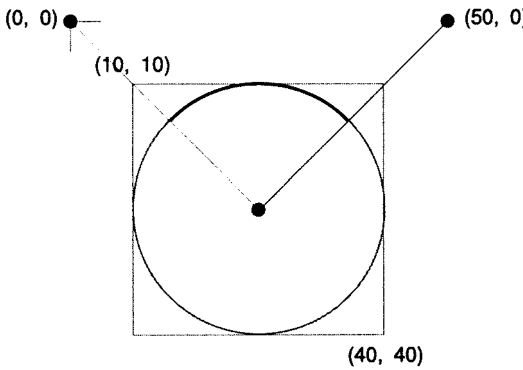
弧

Arc 関数は，指定した長方形に内接する楕円形の外辺部に沿った弧を描きます。矩形枠の中心と指定した開始点とを結ぶ直線が楕円の輪郭線と交わる点を始点として，楕円の中心と指定した終了点とを結ぶ直線が楕円の輪郭線と交わる点まで反時計まわりに線が引かれます。

以下に示すメンバ関数 ***Paint*** は，矩形枠を（10, 10）と（40, 40）で指定し，弧の始点を（0, 0），終点を（50, 0）と指定してハイライト表示の弧を描きます。

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo) {
  Arc(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
}
```

図 18.6
Arc 関数の結果



Arc は現在位置を移動させません。

図形の描画

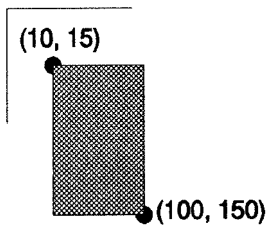
図形を描画する GDI 関数は，ディスプレイコンテキストで現在設定されているペンを用いて輪郭を描き，現在設定されているブラシでその輪郭の中を塗りつぶします。現在位置には影響しません。

矩形

Rectangle は，左上隅と右下隅を指定して長方形を描きます。たとえば，メンバ関数 ***Paint*** に以下の文を書けば，(10, 15) から (100, 150) を頂点とする長方形が描かれます。

```
Rectangle(PaintDC, 10, 15, 100, 150);
```

図 18.7
Rectangle 関数の結果

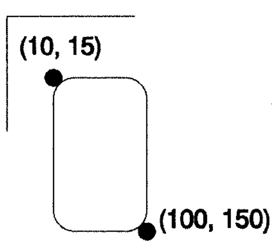


面取り矩形

RoundRect は，角の丸い長方形を描きます。丸い角は，楕円の 4 分の 1 として定義されます。たとえば，メンバ関数 ***Paint*** に以下の文を書くと，(10, 15) から (100, 150) を頂点とする長方形を描きますが，各頂点は，幅 9，高さ 11 の楕円を 4 分の 1 ずつ使ったものとなります。

```
RoundRect(PaintDC, 10, 15, 100, 150, 9, 11);
```

図 18.8
RoundRect 関数の結果

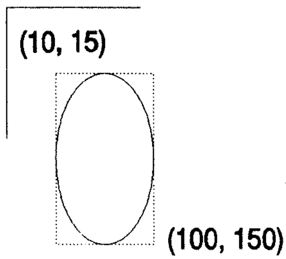


楕円

Ellipse は，矩形枠によって定義される楕円を描きます。以下に示す例では，(10，15）と（100，150）を頂点とする長方形に内接する楕円を描きます。

```
Ellipse(PaintDC, 10, 50, 100, 150);
```

図 18.9
Ellipse 関数の結果

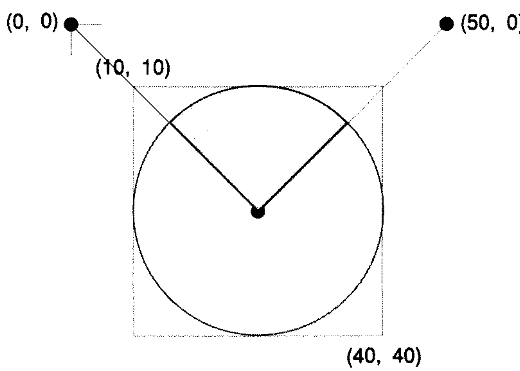


扇型と弦

Pie と *Chord* は，楕円の断片を描きます。どちらも *Arc* 関数と同じように弧を描きます。しかし，*Pie* と *Chord* は線ではなく図形を描きます。*Pie* 関数は，楕円の中心と弧の端点とを結びます。以下に示す例では，*Pie* 関数により (10, 10）と（40，40）を頂点とする長方形に内接する円の上 4 分の 1 を描きます。

```
Pie(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

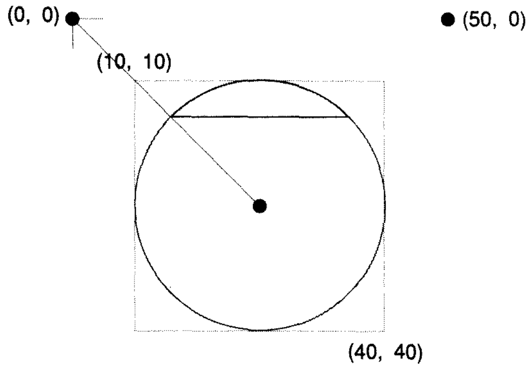
図 18.10
Pie 関数の結果



Chord 関数は，弧の２つの端点を結びます。

```
Chord(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

図 18.11
Chord 関数の結果



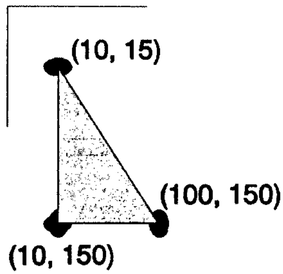
多角形

Polygon では，***Polyline*** 関数と同様に連続する線分を描きますが，最後に指定した点と最初に指定した点を線分で結んで閉じられた図形を描くという点が異なります。図形を描いてから，現在のブラシで多角形の中を塗りつぶします。以下に示すメンバ関数 ***Paint*** は直角三角形を塗りつぶします。

```
void TMyWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
POINT Points[3];

  Points[0].x = 10;
  Points[0].y = 15;
  Points[1].x = 10;
  Points[1].y = 150;
  Points[2].x = 100;
  Points[2].y = 150;
  Polyline(PaintDC, Points, 3);
}
```

図 18.12
Polygon 関数の結果



パレットの使い方

ディスプレイ装置によっては，たくさんの色を表示可能ではあっても，同時に表示できる色数が限られているものがあります。このように，現在ディスプレイに同時に表示できる色の組み合わせのことを，システムパレットまたは物理パレットと呼びます。Windows では，ディスプレイ装置のシステムパレットにどんな色を組み入れるかという点について，ある程度アプリケーション側から制御できるようになっています。アプリケーションで単純な色しか用いない場合は，パレットを操作する必要はありません。

しかし，システムパレットを操作してしまうと，画面上に描かれるすべてのものに影響が出てしまいます。他のアプリケーションが描いたものも例外ではありません。1つのアプリケーションを変更すると，他のすべてのアプリケーションの色が狂ってしまうこともあるのです。そこで，Windows のパレットマネージャが，システムパレットを変更しようとするアプリケーション間の仲立ちをすることによってこの問題を回避しています。Windows は，各アプリケーションに対し論理パレットを提供します。この論理パレットには，アプリケーションで必要な色の組み合わせが収められます。パレットマネージャは，この論理パレットで要求されている色をシステムパレット内の色に対応づけます。要求されている色がシステムパレット内にない場合は，パレットマネージャがシステムパレットにその色を追加します。システムパレットに追加する色を入れる余地がない場合，この色はシステムパレット内の色のうち，最も近い色に対応づけられます。

アプリケーションがアクティブになった時点で，論理パレットの色をシステムパレットに入れることができます。この結果，別のアプリケーションの論理パレットで指定した色が強制的に除外される場合があります。どんな場合でも，

Windowsは20個の固定色をシステムパレットに予約してあり，アプリケーションごとの配色やWindows自体の配色を大体は保存できるようになっています。

パレットの設定

「**描画ツール**」の節で説明したペンやブラシのように，論理パレットも描画ツールです。論理パレットを生成するには ***CreatePalette*** を使います。***CreatePalette*** は，論理パレットへのハンドルを返してくるので，このハンドルを ***SelectPalette*** に渡せば，論理パレットをディスプレイコンテキストに組み込むことができます。そこで，***LOGPALETTE*** 構造体を指すポインタを ***CREATEPALETTE*** に渡します。この ***LOGPALETTE*** 構造体には，ヘッダ情報とパレットエントリの配列が保持されます。パレットの各エントリは ***PALETTEENTRY*** という構造体で，***peRed***，***peGreen***，***peBlue*** という色指定用メンバが3つと，***peFlags*** というフラグ用メンバが1つあります。次の ***GetStockObject*** で，システムパレットの20色の常備色が入ったデフォルトパレットが作成されます。

```
GetStockObject(DEFAULT_PALETTE);
```

アプリケーションが ***SelectPalette*** を用いてパレットをディスプレイコンテキストに組み込んだら，そのパレットを使う前に，そのパレットをアプリケーションに認識させる必要があります。次のようにWindows関数 ***RealizePalette*** を使ってください。

```
ThePalette = CreatePalette(&ALogPalette);
SelectPalette(TheDC, ThePalette, 0);
RealizePalette(TheDC);
```

RealizePalette 関数を呼び出すと，アプリケーションの論理パレットの色が表示装置のシステムパレットに組み込まれます。Windowsは，論理パレットの色についてまずシステムパレットにすでに存在する色と一致するものを対応づけ，システムパレットにない色については，追加する余地があればシステムパレットに追加します。色を追加する余地がなくなり，システムパレットにも完全に一致する色がない場合には，最も似た色に対応づけます。

アプリケーションは描画を開始する前に描画ツールを選択するように，パレットも準備する必要があります。

パレットを用いた描画

アプリケーションのパレットが準備されたら，パレットの色を使って描画できるようになります。パレットの色は直接的にでも間接的にでも指定できます。パレットを直接指定するには，パレットインデックス ***COLORREF*** を使用します。パレットインデックス ***COLORREF*** は ***DWORD*** 値で，最上位バイトを1に設定し，論理パレットの番号を下位2バイトに設定します。たとえば，0x01000009は，論理パレットの9番目の色を指定することを意味します。この数値は ***COLORREF*** を引数としてとる場合であれば，いつでも使うことができます。たとえば以下のような使い方が可能です。

```
ALogPen.lopnColor = 0x01000009;
```

表示装置にシステムパレットがなく，24ビットカラーをすべて表示可能である場合，パレットインデックスを使うと使える色の種類が論理パレットの色に限定されてしまいます。このような制限を回避するには，パレット相対の ***COLORREF*** 値を使ってパレットカラーを間接的に指定する方法があります。パレット相対の ***COLORREF*** は，RGB値を直接示す ***COLORREF*** と同じものですが，最上位バイトが2である点が異なります。下位3バイトはRGB色の値を保持します。たとえば0x020000FFは，純粋な赤を表わすパレット相対 ***COLORREF*** 値を示します。表示装置がシステムパレットをサポートしている場合，Windowsは選択された論理パレットの色に最も近い色にRGB情報を合わせます。表示装置がシステムパレットをサポートしていない場合，WindowsはRGB値が直接指定されたものとして ***COLORREF*** を使用します。

パレット情報の取得

GetPaletteEntries を呼び出せば，パレットに関する情報を取得できます。この関数は，指定された ***PALETTEENTRY*** 構造体配列にパレットのエントリを登録します。最初のパレットエントリのインデックスとエントリ数も ***GetPaletteEntries*** を呼び出すときに指定できます。

パレットの変更

SetPaletteEntries か ***AnimatePalette*** を呼び出せば，論理パレット内の色を変えられます。***SetPaletteEntries*** は ***GetPaletteEntries*** と同じ引数を取り，指定された色を第3引数で指定された色に変更します。なお，この変更は ***RealizePalette*** が呼び出されるまではシステムパレットに反映されず，またクライアント領域が再描画されるまでは表示もされません。***AnimatePalette*** は ***SetPaletteEntries*** と同じ引数を取ります。***AnimatePalette*** はアプリケーションでパレットを変更し，その変更を即座に表示に反映させたい場合に使います。***AnimatePalette*** を呼び出すと，パレット内の色のうち，***peFlags*** フィールドが ***PC_RESERVED*** 定数に設定されている色は，その番号に対応する新しい色に置き換えられ，すぐにシステムパレットに対応づけられます。それ以外の色については変更されません。

たとえば，パレットの先頭の10色分を取り出し，個々の色の赤の成分を増加させ，青と緑の成分を減らし，しかもその変更をすぐにウィンドウに反映させたい場合を考えてみます（パレット内の一部の色には ***PC_RESERVED*** が設定されているものとします)。以下のようにして行ないます。

```
GetObject(ThePalette, sizeof (NumEntries), &NumEntries);
if (NumEntries >= 10) {
  GetPaletteEntries(ThePalette, 0, 10, &PaletteEntries);
  for (int i = 0; i <= 9; ++i)
  {
    PaletteEntries[i].peRed = PaletteEntries[i].peRed + 40;
    PaletteEntries[i].peGreen = PaletteEntries[i].peGreen - 40;
    PaletteEntries[i].peBlue = PaletteEntries[i].peBlue - 40;
  }
  AnimatePalette(ThePalette, 0, 10, &PaletteEntries);
}
```

AnimatePalette 関数を使わずに同じことを実現するには，まず以下のコードを実行し，

```
SetPaletteEntries(ThePalette, 0, 10, &PaletteEntries);
RealizePalette(ThePalette);
```

そのあとに，ウィンドウを再描画して色の変化をウィンドウに反映させます。

パレット変更に対する応答

アクティブウィンドウが論理パレットを対応づけ，システムパレットを変更すると，他のウィンドウに ***WM_PALETTECHANGED*** メッセージが送られます。このメッセージを受け取ったウィンドウは，以下の3つの方法のどれかで応答します。

- 何もしない（色が狂う可能性があります）。
- 論理パレットをシステムパレットに対応させ，ウィンドウを再描画する（反応は遅くなりますが，色は最も正しいものになります）。
- 論理パレットをシステムパレットに対応させ，その後で ***UpdateColors*** を呼び出し即座にクライアント領域をシステムパレットに合わせて更新する。

UpdateColors を使う方法は，クライアント領域を再描画するより一般的には速いのですが，色の正確さが若干損なわれる可能性があります。

WM_PALETTECHANGED メッセージの ***WParam*** フィールドには，論理パレットをシステムパレットに対応づけたウィンドウのハンドルが収められています。このメッセージに対する処置として論理パレットをシステムパレットに対応し直す場合，無限ループにならないように，メッセージを発信したウィンドウのハンドルが自分のウィンドウへのハンドルではないことをまず確かめる必要があります。

マスターディスクの EXAMPLES ディレクトリに入っているアプリケーション PALTEST.CPP では，8色の論理パレットを生成し，システムパレットへ対応づけています。マウスの左ボタンをクリックすると，各々の色で正方形が描かれます。マウスの右ボタンをクリックすると，論理パレット内で8色が循環移動します。パレットインデックス ***COLORREF*** が使われているので，論理パレットが変更され，箱型が再描画されると，箱の色も変化します。パレットインデックス ***COLORREF*** を作成するには，***PALETTEINDEX*** マクロを使います。

第19章

リソースの詳細

Winows上の大部分のプログラムは，標準的なユーザインターフェースによって操作性の高いものになっています。たとえば，Windows上のプログラムはほとんどがメニューを選択するだけでプログラムのコマンドを実行でき，またマウスポインタを表わすカーソルの形状を変えて，矢印やペイントブラシなどの多種多様なツールに対応させることができます。

Windowsプログラムのリソースの例としてメニューとカーソルが挙げられます。リソースは，プログラムの通常のデータセグメントとは別に実行形式(.EXE)ファイルに格納されるデータです。リソースはプログラムのコードとは独立して設計，規定されます。最終的には，コンパイルしたプログラムコードにリソースを追加して，プログラムの実行形式ファイルを生成します。

最も頻繁に作成，利用するリソースとしては以下のものがあります。

- メニュー
- ダイアログボックス
- アイコン
- カーソル
- キーボードアクセラレータ
- ビットマップ
- 文字列

通常，Windowsはアプリケーションをメモリ内に読み込むとき，リソースをディスク上に残しておきます。プログラムを実行していてリソースが必要になると，個々のリソースを読み込みます。ビットマップ以外のリソースは不要になるとメモリから廃棄されます。リソースの属性を変えることによって，プログラムを読み込むときにリソースも一緒に読み込まれるようにしたり，Windowsがメモリからリソースを廃棄しないようにしたりできます。

リソースの作成

リソースは，リソースエディタやリソースコンパイラで作成します。Borland C++では，どちらの方法でも作成できるので，目的のリソース作成に適した方法を使えます。たいていは，リソースエディタを使って実際にリソースの外観を見ながら作成する方が簡単です。また，ときには本や雑誌に載っているリソーススクリプトファイルをリソースコンパイラでコンパイルする方が便利なこともあるでしょう。

どちらの方法を用いるにせよ，通常はアプリケーションごとにリソースファイル（.RES）を作成します。このリソースファイルには，アプリケーションで使われるメニュー，ダイアログボックス，ビットマップなどの全リソースに関する情報がバイナリ形式で収められています。

この章で説明するように，バイナリ形式のリソースファイル（.RES）は，リソースコンパイラを使って実行形式ファイルに追加します。さらに，リソースをメモリに読み込むためのコードもプログラムに書いておく必要があります。各リソースはメモリ内に個別に読み込まれます。プログラムは現在使用中のリソース用のメモリしか必要としないので，メモリは効率的に利用されます。リソースのメモリへの読み込み方法についてもこの章で説明します。

実行形式ファイルへのリソースの追加

バイナリ形式の.RESファイルに格納されたリソースを，プログラムの実行形式ファイル（.EXEファイル）に結合してください。こうして，コンパイル済みのアプリケーションコードとリソースの両方を含むファイルができるわけです。

リソースを実行形式ファイルへ追加するには，以下の2つの方法があります。

- リソースエディタを用いて，.RESファイルからコンパイル済みの.EXEファイルにリソースを複写します。
- 次の節で説明するように，リソースコンパイラ（RC）を用いて，.RESファイルからコンパイル済みの.EXEファイルにリソースを複写します。

リソースコンパイラの実行

この節では，Microsoft 社のリソースコンパイラを使ってリソースファイルを生成する方法を簡単に説明します。リソースコンパイラは，リソーススクリプトファイルを入力として受け取り，バイナリ形式のリソースファイルを生成するDOS 上のアプリケーションです。リソースコンパイラで，リソースを実行形式ファイルに直接追加することもできます。

リソースコンパイラは，DOS のプロンプトか Windows の DOS ウィンドウで動作します。コマンドラインの書式は以下のようになります。大カッコで囲まれている項目はオプションです。

RC [オプション]〈入力.RC ファイル名〉[出力.EXE ファイル名]

使用例を示します。

```
RC SAMPPROG.RC
```

このコマンドを実行すると，スクリプトファイル SAMPPROG.RC を読み込んで SAMPPROG.RES ファイルを作成し，これを SAMPPROG.EXE ファイルに直接追加します。SAMPPROG.EXE 内の既存のリソースはすべて新しいものに置き換えられます。

どちらの方法でも，C++プログラム SAMPPROG.CPP にリソースをメモリに読み込むための命令を記述する必要があります。

表 19.1 はリソースコンパイラで有効なコマンドラインオプションの一覧です。

表 19.1 リソースコンパイラのコマンドラインオプション

オプション	意味
-r	.RES ファイルの作成
-l	LIM 3.2 EMS を使用するアプリケーションの作成
-e	EMS メモリを使用する DLL の作成
-m	マルチインスタンスフラグの設定
-p	プライベートライブラリの作成
-t	プロテクトモード専用アプリケーションの作成
-v	詳細出力モード（コンパイル過程メッセージを表示）
-d	プリプロセッサによるシンボルの定義
-fo	出力する.RES ファイル名の変更
-fe	出力する.EXE ファイル名の変更
-i	INCLUDE 文の検索パスの指定
-x	INCLUDE 環境変数を無視
-k	.DEF ファイルで指定した順にセグメントを配置（この指定がなければ，リソースを高速に読み込めるようにセグメントをソートします）。

リソースコンパイラスクリプトファイル

リソースコンパイラが読み込むスクリプトファイル（.RC ファイル）はリソースを定義したテキストファイルで，このファイルをコンパイルしてバイナリ形式のリソースファイルを得ます。リソースコンパイラの書式によるリソースの定義は，プログラムの作成とよく似ています。ここでは，リソースコンパイラのスクリプト言語を詳細に説明するかわりに，サンプルをいくつか用いて，スクリプトファイルのコンパイルや編集時に必要な知識を示します。ただし，複雑なリソースを作成する場合はたいていリソースエディタを用いる方が簡単です。また，必要ならリソースエディタで直接スクリプトファイルを生成することもできます。

以下に，リソーススクリプトファイルの例（SAMPPROG.RC）を示します。

```
; SAMPROG.RC --リソーススクリプトファイルの例
;             --このスクリプトファイルでは、Windowsにおけるアイコン、文字列、
;               メニュー、ダイアログの各リソースの使用方法を示す
;             --このスクリプトファイルをリソースコンパイラでコンパイルし、
;               C++プログラムに追加する
; 定数定義には、ヘッダファイルをインクルードする：
#include <windows.h>
#include <owlrc.h>
#include <sampprog.h>

; アイコンは別ファイルで定義されている
SampleIcon ICON sample.ico

; 文字列は番号とシンボルのどちらでも定義できる
STRINGTABLE
BEGIN
  IDS_NOPROGRM, "Program unavailable"
  IDS_INVALID, "Invalid entry"
  IDS_FATAL, "Fatal error!"
; .
; . 他の文字列をここに記述する
; .
END

; メニューでユーザが使用するコマンドを定義する
SampleMenu MENU
BEGIN
  POPUP "&File"
    BEGIN
     MENUITEM "&New..¥tCtrl+N",          CM_FILENEW
     MENUITEM "&Open..¥tCtrl+O",         CM_FILEOPEN
     MENUITEM "&Save..¥tCtrl+S",         CM_FILESAVE
     MENUITEM SEPARATOR
     MENUITEM "&Print ¥tCtrl+P",         CM_FILEPRINT
    END
   POPUP "&Edit"
    BEGIN
     MENUITEM "&Undo ¥tAlt+BkSp",        CM_EDITUNDO
     MENUITEM SEPARATOR
     MENUITEM "&Cut ¥tShift+Del",        CM_EDITCUT
     MENUITEM "C&opy ¥tIns",             CM_EDITCOPY
     MENUITEM "&Paste ¥tShift+Ins",      CM_EDITPASTE
     MENUITEM "&Delete ¥tDel",           CM_EDITDELETE
     MENUITEM "&C&lear All ¥tCtrl+Del", CM_EDITCLEAR
     END
   POPUP "&View"
     BEGIN
      MENUITEM "Summary¥tF2",           CM_VIEWSUMMARY
      MENUITEM "Graph¥tF3",             CM_VIEWGRAPH
     END
 END
```

```
; アクセラレータは、メニューやコマンドへのホットキーとして使われる
SampleAccelerators ACCELERATORS
BEGIN
  "^n", CM_FILENEW
  "^o", CM_FILEOPEN
  "^s", CM_FILESAVE
  "^p", CM_FILEPRINT
  VK_DELETE, CM_EDITCUT, VIRTKEY, SHIFT
  VK_INSERT, CM_EDITCOPY, VIRTKEY
  VK_INSERT, CM_EDITPASTE, VIRTKEY, SHIFT
  VK_DELETE, CM_EDITDELETE, VIRTKEY
  VK_DELETE, CM_EDITCLEAR, VIRTKEY, CONTROL
  VK_BACK, CM_EDITUNDO, VIRTKEY, ALT
  VK_F2, CM_VIEWSUMMARY, VIRTKEY
  VK_F3, CM_VIEWGRAPH, VIRTKEY
END

; ダイアログボックスにはユーザに対する各種情報が表示される
AboutBox  DIALOG 20, 20, 160, 80
CAPTION "About SAMPPROG"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  CTEXT "Sample Program"           -1,  0,  10, 160, 10
  CTEXT "Written in ObjectWindows" -1,  0,  26, 160, 10
  CTEXT "for Windows 3.0"          -1,  0,  36, 160, 10
  ICON "SAMPLE_ICON"               -1,  8,   8,   0,  0
  DEFPUSHBUTTON "OK",            IDOK, 60,  56,  40, 14
END

FileOpen DIALOG 20, 20, 204, 124
DAPTION "File Open"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  LTEXT    "File&name:",     -1,   6,   8,  36,  10
  EDITTEXT                  100,  42,   6,  98,  12, ES_AUTOHSCROLL
  LTEXT    "Directory:",     -1,   6,  20,  36,  10
  LTEXT    ""               101,  42,  20,  98,  10
  LTEXT    "&Files:",        -1,   6,  32,  64,  10
  LISTBOX  102, 6, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  LTEXT    "&Directories:", -1,  76,  32,  64,  10
  LISTBOX  103, 76, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  DEFPUSHBUTTON "OK",          IDOK, 146,  6,  50,  14
  PUSHBUTTON    "Cancel", IDCANCEL, 146, 24,  50,  14
END
```

ヘッダファイル SAMPPROG.H に C 言語形式で定数定義を記述します。ヘッダファイルの典型例を以下に示します。

```
/* SAMPPROG.H -- SAMPPROG.RC 用の定数定義用ヘッダファイル
#define IDS_NOPROGRM       601
#define IDS_INVALID        602
#define IDS_FATAL          603

#define CM_FILENEW         701
#define CM_FILESAVE        702
/*

  .
  . 他の定数をここに記述する
  .
*/
```

リソースコンパイラ使用上の注意

リソースコンパイラのスクリプトプログラミング言語について注意すべき点をいくつか示します。

1. セミコロン（;）以降行末までおよび/*と*/ではさまれたテキストはコメントとして扱われます。
2. BEGIN や END などのキーワードについては**大文字小文字を区別しません。**
3. ***CM_FILENEW*** や ***AboutBox*** などのシンボル名については大文字小文字を区別します。
4. 空白は自由に使用できます。
5. **#define** で定義した定数のヘッダファイルは **#include** 指令を用いてインクルードしなければなりません。
6. **rcinclude** 指令を用いて，別のリソーススクリプトファイル（*.RC または *.DLG）をインクルードできます。たとえば，ObjectWindows ファイルダイアログボックスをインクルードできます。
7. メニュー項目 ID および文字列 ID はそれぞれ重複があってはいけません。
8. メニューをネストできます。
9. メニュー項目やダイアログコントロール項目中のアンパサンド（&）は，プログラム実行時に&の次の文字に下線が引かれることを示します。この下線のついたメニュー項目やダイアログコントロール項目は，***GRPH*** キーと下線つきの文字を同時に押すことにより選択できます。

10. メニュー項目中の¥t文字はタブ文字に展開され，文字列の整列に利用されます。メニュー項目に対応するキーボードコマンドの整列表示に多用されます。
11. ホットキーは，アクセラレータテーブルで定義できます。
12. スタティックテキストのように，使われてもプログラムの流れが変わらないコントロール項目のID番号は通常−1とします。
13. ダイアログやコントロールの位置は，左端位置，上端位置，水平幅，高さの順で指定します。
14. メニュー項目，ダイアログ，およびスクロールバー，エディットフィールド，ボタンなどのダイアログのコントロール項目の形状は，Windows形式の定数で変更できます。ビット演算子**or**（|）を用いて形状属性を組み合わせてください。たとえば，コントロール項目のスタイル属性として***WS_CHILD***，***WS_TABSTOP***，***WS_BORDER***を持たせるには，***WS_CHILD*** | ***WS_TABSTOP*** | ***WS_BORDER*** |で指定します。
15. たいていは，スクリプトファイルで複雑なリソース定義を行なうよりも，リソースエディタを使う方が簡単です。
16. リソースを読み込んで利用するためのコードをアプリケーションに含める必要があります。

アプリケーションへのリソースの読み込み

実行形式ファイルにリソースを追加したのち，明示的にアプリケーションにリソースを読み込むと，リソースが利用できるようになります。リソースの読み込み方法はリソースの種類によって異なります。

メニューリソースの読み込み

ウィンドウのメニューは，ウィンドウの作成属性の1つです。つまり，ウィンドウ自身がメンバ関数 ***Create*** によって作成される前に，ウィンドウ自体の特性が指定されていなければなりません。したがって，ウィンドウクラスのコンストラクタ内か，または構築した直後にメニューを指定します。新しいウィンドウオブジェクトが生成されると，メニューID文字列を引数としてメンバ関数 ***AssignMenu*** が呼び出され，メニューリソースが読み込まれます。メンバ関数 ***AssignMenu*** は ***TWindow*** から継承しています。以下に例を示します。

```
SampleMainWindow::SampleMainWindow(PTWindowsObject AParent,
  LPSTR ATitle) : TWindow(AParent, ATitle)
{
  AssignMenu(100);
:
}
```

上の例では，***AssignMenu*** 関数でID番号100のメニューリソースを新しいウィンドウオブジェクトに読み込みます。リソース指定には，数値識別子以外に“SampleMenu”などのシンボル名（文字列）を使うこともできます。シンボル名を使うと，上のコードは以下のようになります。

```
SampleMainWindow::SampleMainWindow(PTWindowsObject AParent,
  LPSTR ATitle) : TWindow(AParent, ATitle)
{
  AssignMenu("SampleMenu");
:
}
```

ウィンドウオブジェクトの生成方法の詳細については，第10章**『ウィンドウオブジェクト』**を参照してください。

メニュー選択の処理は，そのメニューが属するウィンドウ用のメンバ関数を定義するだけで行なえます。以下のように ***CM_FIRST*** 識別子のついた専用のメンバ関数定義構文を使ってください。

```
virtual void HandleMenu101(RTMessage Msg) = [CM_FIRST + 101];
```

101は，選択されたメニュー項目のIDです。メニューリソースのIDではありません。各メニュー項目には，それぞれ互いに重複しない整数値のID番号があります。このメンバ関数で，選択されたメニュー項目の作業を呼び出すようにします。通常は，メニューコマンドに対応したシンボル名によって定数を定義します。

アクセラレータテーブルの読み込み

アクセラレータとは，ホットキーまたはキーボードショートカットとも呼ばれ，アプリケーションコマンドの指定に使います。通常は，メニュー選択にかわる方法としてアクセラレータを定義します。たとえば，Editメニューで Deleteを選択するかわりに，***DEL*** キーが標準アクセラレータとして定義されています。また，アクセラレータをメニュー項目と対応しないコマンドに割り当てることもできます。

アクセラレータリソースは，アクセラレータテーブルに格納されます。アクセラレータテーブルの読み込みにはWindows関数 ***LoadAccelerators*** を使ってください。この関数は，このテーブルのハンドルを返します。メニューリソースでは対応するウィンドウが1つに限定されますが，アクセラレータリソースはアプリケーション全体に対応します。アプリケーションは，アクセラレータテーブルを一度に1つしか設定できません。アプリケーションオブジェクトには，アクセラレータリソースのハンドルが入るデータメンバ ***HAccTable*** があります。以下に示すように，アクセラレータリソースの読み込みは，通常アプリケーションオブジェクトのメンバ関数 ***InitInstance*** で行ないます。

```
void SampleApplication::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(GetApplication()->HInstance,
                               "SampleAccelerators");
}
```

アクセラレータキーの多くはメニュー選択のショートカットとして定義されます。たとえば，通常 ***SHIFT-INS*** キーは貼り付けのショートカットとして使われます。アクセラレータを選択すると，メニュー選択によって生成されるメッセージと同一のコマンドメッセージが生成されます。メニュー選択に対応するメンバ関数をアクセラレータキーに対応させるには，リソース内で定義されるアクセラレータの値をメニュー項目のID番号に一致させます。

ダイアログボックスの読み込み

ダイアログボックスは，ObjectWindowsのクラスに直接対応する唯一のリソース型です。***TDialog***とその派生クラスには，ダイアログボックスリソースを利用するためのインターフェースオブジェクトが定義されています。ダイアログボックスオブジェクトにはそれぞれ1つのダイアログボックスリソースが対応し，このリソースによってダイアログボックスの大きさや位置，ボタンやリストボックスなどのコントロールの構成が規定されます。

ダイアログボックスの生成時にダイアログボックスのリソースを指定します。メニューリソースやアクセラレータリソースと同様，ダイアログリソースの指定にはシンボル名か整数値IDのどちらも使用できます。たとえば，

```
ADlg = new TSampleDialog(this, "AboutBox");
```

という書式も，

```
ADlg = new TSampleDialog(this, 120);
```

という書式も使用できます。

ダイアログオブジェクトの作成については，第5章『**ダイアログの追加**』を参照してください。

カーソルとアイコンの読み込み

各ウィンドウクラスには，**登録属性**と呼ばれる特別な属性があります。登録属性には，ウィンドウのカーソルやアイコンに関するものも含まれます。ウィンドウクラスに対してこの属性を設定するには，***GetWindowsClass***または***GetClassName***というメンバ関数を定義します。

たとえば，リストボックスの項目の選択時に用いるカーソルを作成したとします。そこで，人差し指の形のカーソルが，‘Finger’というカーソルリソース名で格納されているとします。さらに，幸せな顔を表示するアイコンリソース‘SampleIcon’を作成してあるとします。この場合，以下に示すようにメンバ関数***GetWindowClass***を記述します。

```
void SampleWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(GetApplication()->HInstance,
                                 "Finger");
  AWndClass.hIcon = LoadIcon(GetApplication()->HInstance,
                             "SampleIcon");
}
```

ただし，カーソルとアイコンには相違点が1つあります。カーソルは1つのウィンドウに対して指定されますが，アイコンはアプリケーション全体に対して定義される点です。このため，アイコンの設定はメインウィンドウのクラスで行ないます。この1アイコン1アプリケーション原則には次の2つの例外があります。

- MDI標準に従うアプリケーションの各MDI子ウィンドウは独自のアイコンを持ちます。
- アプリケーションの各ポップアップウィンドウは独自のアイコンを持ちます。

Windowsが提供するストックカーソルやストックアイコンを利用するには，*HInstance*のかわりにヌルを渡し，ID番号としてカーソルの場合は***IDC_IBEAM***などの***IDC_***で始まる定数を，アイコンの場合は***IDI_HAND***などの***IDI_***で始まる定数を使います。以下に例を示します。

```
void SampleWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClas(AWndClass);
  AWndClass.hCursor = LoadCursor(NULL, IDC_IBEAM);
  AWndClass.hIcon = LoadIcon(NULL, IDI_HAND);
}
```

ウィンドウ登録属性の詳細については第10章**『ウィンドウオブジェクト』**を参照してください。

文字列リソースの読み込み

アプリケーションの文字列をリソースとして定義する主な理由は，特定の用途へのアプリケーションのカスタマイズや，他国語へのアプリケーションの移植を容易にするためです。文字列がソースコード内で定義されていると，文字列の変更や翻訳を行なうときにソースコードそのものを改変する必要があります。一方，文字列がリソースとして定義されている場合は，アプリケーションの実行形式ファイルの文字列テーブルに文字列が格納されるので，文字列エディタで文字列テーブル内の文字列を変更できます。しかも，ソースコードは変更どころか参照する必要すらありません。実行形式ファイルに格納できる文字列テーブルは1つだけです。

文字列テーブルからアプリケーションのデータセグメント内にあるバッファに文字列を読み込むには ***LoadString*** 関数を用います。***LoadString*** 関数呼び出しの書式は以下の通りです。

```
LoadString(GetApplication()->HInstance, StringID, TextItem,
                    MaxChars);
```

- 引数 *StringID* で文字列テーブル内の文字列のID番号（601など）を指定します。数値のかわりに定数シンボルも使用できます。
- 引数 *TextItem* でバッファを指す ***LPSTR*** 型ポインタを指定します。
- 引数 *MaxChars* でバッファに転送する最大文字数を指定します。文字列リソースの最大長は255文字なので，文字列全体を確実に取得するには，バッファ長を256文字以上にします。

LoadString 関数は，バッファに転送した文字列数を返します。指定したリソースが存在しない場合には，0を返します。

メッセージボックス内の文字列表示に文字列リソースを使用できます。たとえば，エラーメッセージの表示に使います。この場合，文字列‘プログラム使用不可’を文字列テーブルに定義し，その文字列のIDとして ***IDS_NOPRGRM*** 定数を定義します。エラーボックスでこの文字列リソースを使うには，以下のような手続きを作成します。

```
virtual void TestDialog::RunErrorBox(int ErrorNumber) {
  char TextItem[255];

  LoadString(GetApplication()->Hinstance, IDS_NOPRGRM, TextItem, 20);
  MessageBox(HWindow, TextItem, "Error", MB_OK | MB_ICONEXCLAMATION);
}
```

この例では，単一の文字列をエラーボックスに表示します。一連の文字列をリストボックスに表示するには，まず ***LoadString*** 関数を呼び出して文字列を 1 つずつ取得し，次に ***AddString*** 関数を呼び出してリストボックスに文字列を追加します。

文字列リソースは，ソースコードで指定したメニューのメニュー項目の作成や追加にも使えます。この場合，まず ***LoadString*** 関数を呼び出して文字列リソースを取得し，次にその文字列を引数として Windows 関数の ***CreateMenu*** か ***AppendMenu*** を呼び出します。以下に例を示します。

```
SampleWindow::SampleWindow(PTWindowsObject AParent,
  LPSTR ATitle) : TWindow(AParent, ATitle)
{
  char TextItem[255];

  AssignMenu[100];
  LoadString(GetApplication()->HInstance, 301, TextItem, 20);
  AppendMenu(GetMenu(HWindow), MF_STRING | MF_ENABLED, 501,
TextItem);
}
```

ビットマップの読み込み

Windows 関数 ***LoadBitmap*** でビットマップリソースを読み込みます。***LoadBitmap*** 関数は，ビットマップをメモリに読み込み，ビットマップへのハンドルを返します。たとえば次のように使います。

```
HMyBit = LoadBitmap(GetApplication()->HInstance,
                    MAKEINTRESOURCE(501));
```

この例では，ID 番号が 501 のビットマップリソースを読み込み，そのビットマップのハンドルを変数 ***HMyBit*** に格納します。ビットマップを読み込むと，明示的に削除しない限りメモリからは除去されません。他のリソースとは異なり，アプリケーションがクローズされてもビットマップはメモリ内に保持されます。

Windowsでは，定義済みのビットマップが多数提供され，Windowsのグラフィックスインターフェースの部品として使われています。アプリケーションでも ***OBM_DNARROW***，***OBM_CLOSE***，***OBM_ZOOM*** などの定義済みビットマップを読み込むことができます。定義済みビットマップを読み込むには，定義済みアイコンや定義済みカーソルと同様，*GetApplication()－＞HInstance* のかわりにヌルを指定して ***LoadBitmap*** 関数を呼び出します。

```
HMyBit = LoadBitmap(NULL, OBM_CLOSE);
```

ビットマップを読み込むと，アプリケーションで以下のような用途に利用できます。

- 画面上への絵の表示。たとえば，アプリケーションの「About」ボックスにビットマップを読み込み，アプリケーションのロゴとして表示できます。
- 画面の特定領域の塗りつぶしやウィンドウのバックグラウンド領域に用いるブラシの作成。たとえば，ビットマップでブラシを作成すると，バックグラウンド領域を縞模様で埋めることもできます。
- メニュー項目の絵による表示。またはリストボックス内の項目の表示。たとえば，メニュー選択時に「矢印」という文字のかわりに矢印の絵を表示することができます。

ビットマップグラフィックスの使用法については第18章『**GDIについて**』を参照してください。

不要になったビットマップはただちにメモリから削除してください。削除しないと，ビットマップが占めるメモリを他のアプリケーションで利用できません。アプリケーションが使い終わった直後にビットマップを削除しない場合でも，少なくともアプリケーション終了までには削除してください。

次のように Windows 関数 ***DeleteObject*** でビットマップをメモリから削除します。

```
if (DeleteObject(HMyBit)) ・・・ // 成功
```

ビットマップを削除すると，ハンドルは無効となり使えなくなります。

ビットマップによるブラシの作成

ビットマップでブラシを作成して，画面の特定領域を塗りつぶすことができます。領域を1色でベタ塗りすることも，パターンで埋めることもできます。

ブラシとして使えるビットマップの最小サイズは，8×8ピクセルです。ただし，大きなビットマップを用いる場合でも，左上隅の8×8ピクセル分だけがブラシとして用いられます。

図19.1のように，特定の領域を縞模様で埋めることにします。まずリソースエディタを用いて，**図19.2**に示すような8×8のサイズの2つの帯からなるパターンを作成します。次にソースコードでビットマップを読み込み，このパターンでブラシを作成します。**図19.1**に示す領域全体を埋めるために，Windowsはブラシを繰り返し複写します。

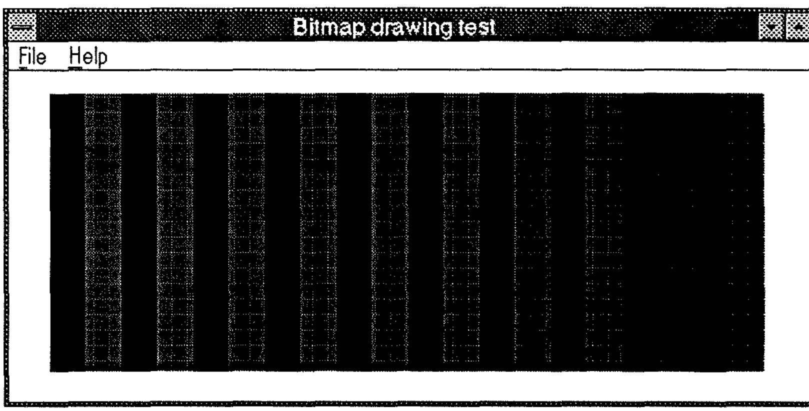
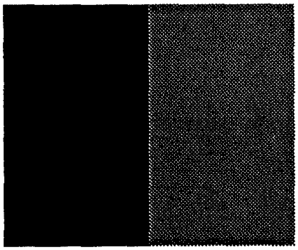


図19.1
画面上の領域を埋める縞模様

ブラシとして用いるビットマップは，実際には8×8ピクセルの大きさしかありませんが，画面全体を埋めることができます。

図 19.2
図 19.1 のパターン用のブラシを作成するビットマップリソース



以下のコードでビットマップパターンをブラシに設定します。

```
void SampleWindow:MakeBrush() {
  LOGBRUSH MyLogBrush;

  HMyBit = LoadBitmap(GetApplication()->HInstance,
                      MAKEINTRESOURCE(502));
  MyLogBrush.lbStyle = BS_PATTERN;
  MyLogBrush.lbHatch = HMyBit;
  TheBrush = CreateBrushIndirect(&MyLogBrush);
}
```

パターンをテストするには，以下に示すようなコードで矩形内にパターンを表示します。

```
void MyWindow:Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  SelectObject(PaintDC, TheBrush);
  Rectangle(PaintDC, 20, 20, 200, 200);
}
```

ブラシを使い終わったら，ビットマップとブラシを削除してください。

```
DeleteObject(HMyBit);
DeleteObject(TheBrush);
```

ビットマップのメニュー内表示

ビットマップをメニューに表示するには ***ModifyMenu*** 関数を用います。この関数は既存のメニュー項目を変更し，メニューエディタで定義したメニュー文字列のかわりにビットマップを表示するようにします。たとえば以下に示すコンストラクタは，ウィンドウにメニューを追加して変更します。

```
SampleWindow::SampleWindow(PTWindowsObject AParent, LPSTR ATitle) :
  TWindow(AParent, ATitle)

  WORD HMyBit; {
  AssignMenu(100);
  HMyBit = LoadBitmap(GetApplication()->HInstance,
                      MakeIntResource(503));
  ModifyMenu(GetMenu(HWindow), 111, MF_BYCOMMAND | MF_BITMAP,
             211, (LPSTR)HMyBit);
}
```

上のコードで，111 は変更対象のメニュー選択肢のコマンド ID，211 は変更後のコマンド ID を示します。変更の前後で同じ ID を使うこともできます。

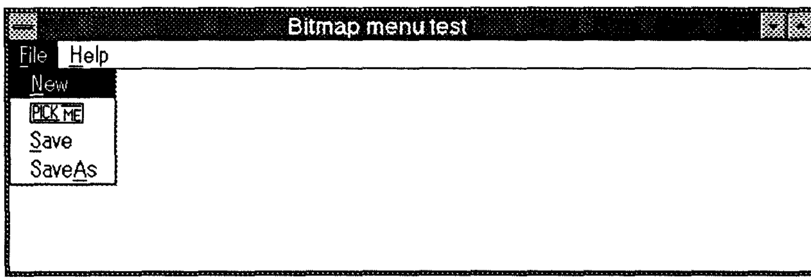


図 19.3
選択肢の 1 つにビットマップを使用したメニュー

第 20 章
アプリケーション作成の指針

Windowsベースのソフトウェアを開発する主な理由として，Windowsアプリケーションは操作が習得しやすく，使いやすい点が挙げられます。各プログラムに共通する操作が同じ方法で利用できるのでWindowsユーザは確実に成果をあげることができます。たとえば，ワープロ，描画アプリケーション，線画アプリケーションでは，Editメニューのメニュー項目とキーボードショートカットキーの操作法が共通しています。つまり，1つのWindowsアプリケーションの使い方をマスターすれば，すべてのWindowsアプリケーションの使い方を習得したことになるのです。

共通操作の利用法に関する公式な指針として，IBMはCommon User Access（CUA）というユーザインターフェース標準を発表しています。CUAのガイドラインは，本来OS/2プレゼンテーションマネージャ用のアプリケーションに対して設定されたものですが，Windowsアプリケーションに対しても適用でき，いわば業界標準規格といえます。この節の内容は，IBMの発表したCUAガイドラインをもとにしています。

設計の原則

アプリケーションをCUAに準拠させる主な目的は2つあります。1つはユーザに操作の結果について確信を持たせること，もう1つは作業の流れをユーザが制御できるようにすることです。

ユーザの期待どおりの反応の提供

ユーザに操作の確信を持たせるためのポイントは，アプリケーションが整合性のとれた信頼性の高い方法で応答することです。以下の点に注意が必要です。

- **メタファーの使用**　ユーザがよく知っている書類などをモデルにすれば，ユーザはアプリケーションの下地になっている概念的なモデルをよく理解することができます。
- **整合性**　アプリケーションの外観とその使用法の整合性をとるようにします。共通した操作には，共通したコマンド（メニューの選択，アイコンのクリックなど）を使用します。似たような作業では画面表示も同じようにします。
- **モードを使わない**　モードとは，ユーザが利用できる機能が限定されている状態のことです。モードを設定すると，ユーザの動作を特定の方向に強いるという悪影響が出ることも少なくありません。しかし，簡単なエラーメッセージの表示や，ツール選択中にモードを設定するのはかまいません。たとえば，カーソルがペンならば，ユーザは線を引くことができますが，図形の移動はできません。ただし，モードを設定する場合は，カーソルの形を変えたり，メニュー項目を淡色表示にするなど，見ただけですぐにモードが判別できるようにしてください。

ユーザによる作業工程の制御

作業工程を指図できると，ユーザは作業内容に自信が持てる上，作業を制御している実感がわくようになります。作業を制御している実感がわけば，コンピュータアプリケーションが得体の知れないものだとは感じなくなり，プログラムが便利な道具と思えるようになります。

- **オプションの一覧表示**　ユーザが選択できる動作を画面に一覧表示するようにします。ツールパレット，カラーパレット，オプションメニューのように，表示や選択の方法にも共通性を持たせます。
- **反応の提供**　ユーザの動作には，必ず表示や音で応答するようにします。

- **試行の支援**　ユーザによるオプションの試行を妨げてはいけません。オプションの試行は，新しいアプリケーションを習得するために欠かせません。Undoオプションも設けるようにします。

標準外観と標準動作

アプリケーションの外観とその動作に関するWindowsの標準規則は，プログラムマネージャや供給されるサンプルプログラムを見れば明らかです。たとえば，アプリケーションはすべてメインウィンドウに出現します。マウスのクリックやダブルクリックは，一般にアプリケーションが変わっても同様に動作します。

一般的な事柄

- **「対象を選択してから動作を指定する」手順の使用**　アプリケーション操作の典型的な手順は以下のようになります。まずユーザは，項目あるいは対象を選択し，次にその対象に対する動作を指定します。たとえばワープロソフトでは，ユーザはまず文字列を選択し，次にフォントの種類やサイズを変更するという手順を踏みます。描画プログラムでは，ユーザはまず矩形を選択し，次に画面上の新しい位置へその矩形を移動するという手順になります。対象の選択や動作の指定には他にもいろいろな方法がありますが，なるべくこの一般モデルに合わせるようにしてください。
- **賢いウィンドウ選択**　Windowsアプリケーションは，複数のウィンドウ上に表示できます。アプリケーションを効率よく使えるように設計するには，いろいろなウィンドウ型とその使い方を熟知する必要があります。**メインウィンドウ**はアプリケーションを統合し，その基盤を提供します。メインウィンドウを最小化してアイコンにするとアプリケーションの動作が停止してしまうので，実際のところメインウィンドウはアプリケーションそのものを表現しているといえます。メインウィンドウの全画面表示，つまり最大化はユーザが指定できます。ポップアップウィンドウやダイアログボックスなどの**二次的なウィンドウ**は，メインウィンドウで行なった動作の結果としてメインウィンドウから起動されます。ポップアップウィンドウやダイアログボックスはそれぞれアプリケーションの個々のモジュールに対応させることができますが，最小化してはいけません。

マウスとキーボードの対応関係

- **選択された対象の表示**　マウスクリックで対象が選択されたときにその対象を矩形で囲んだり対象の色や太さを変えると，ユーザに選択内容を視覚的にわからせることができます。選択マークがその対象に永久に影響してはいけませんが，ユーザがその対象に動作を加えたあとも選択状態が継続するようにします。こうすれば，ユーザは同じ対象に連続して操作を加えることができます。ユーザが別の対象を選択したら，選択状態を解除します。
- **マウスポインタの移動制限の禁止**　マウスポインタは，ウィンドウ間，アプリケーション間を自由に移動できるようにします。マウスポインタがマウスの動作に追従しないように移動を制限したり，強制的に移動させたりしないでください。マウスポインタとマウスとの物理的対応が壊れる恐れがあります。
- **キーボード選択カーソルの使用**　キーボード選択カーソルは，キーボード入力が反映される点を示します。たとえば，ワープロソフトではキーボード選択カーソルを点滅するキャレットで表わしています。キーボード選択カーソルはマウスポインタとは独立して働きます。ただし，マウスをクリックすると両者は同じ位置になります。
- **入力フォーカスの明示**　**入力フォーカス**とは，キーボードやマウスの入力が反映される画面上の場所のことです。たとえば，ユーザがドロップダウンメニューを選択すると，そのメニューが入力フォーカスとなります。また，ユーザが文字列を選択してハイライト表示にすると，その文字列が入力フォーカスとなります。入力フォーカスとなる対象はすぐ識別できるようにしてください。たとえば，選択されたメニュー項目は普通反転色で表示されます。また，ダイアログボックスのコントロールは，選択されると点線で囲まれます。ユーザがマウスをドラッグして一群の項目を選択すると，ドラッグした領域は，ドラッグに対応してリアルタイムで動く点線（**マーキー**）で囲まれます。
- **キーボードインターフェースの設計**　マウスを持っていないユーザや，マウスを使用しないユーザのために，矢印キー，タブキー，キーボードアクセラレータなどによるマウス以外の入力方法を提供します。
- **マウス操作規則への対応**　Windowsには，強制はしていませんがマウス操作規則があります。一般に，マウスの左ボタンを対象の選択や操作に用い，右ボタンを別の作業，たとえばモード変更やオプション一覧のポップアップ表示などに用います。マウスの左ボタンは，以下の規則に従うようにしてください。

- **クリック**は，対象を選択します。
- **ダブルクリック**は，対象を選択し，さらにデフォルトの動作，または対象に関連する動作を起動します。
- **クリック—ドラッグ**は，対象のサイズと位置の変更，文字列の選択と高輝度表示を行ないます。
- ***CTRL*—クリック**は，すでに選択されている対象群へ新たに選択した対象を追加します。
- ***SHIFT*—クリック**は，すでに選択されている対象（アンカー）と新たにクリックした対象との間にある対象群をすべて選択します（Windows のファイルマネージャを参照してください）。

メニュー

Windows は，プルダウンメニューの外観について非常に厳密な標準を提供し，また強制しています。しかし，メニューのスタイルと実装するオプションについては制御の余地が残されています。その際に留意すべき規約やスタイルを以下に示します。

- 最上位のメニュー項目やドロップダウンメニューの項目をキーボードから利用するときは，特定の命令を使います。
- 選択されるとダイアログボックスが開くメニュー項目には，項目文字列のあとに省略符号（…）をつけます。
- アクティブではないメニュー項目は淡色で表示します。

グラフィックス，ビジネス，教育，科学，技術系などほとんどのアプリケーションでは，ファイル操作時，編集操作時，ヘルプシステムの利用時にそれぞれ一連の共通操作を行ないます。このことから，CUA では，File メニュー，Edit メニュー，Help メニューに特定の書式を定めています。さらに，View メニューと Options メニューにも指針を規定しています。CUA では，最上位のメニュー項目として以下のような順番で配置するよう定めています。すなわち，Help メニューは，他にメニュー項目がいくつある場合でも**必ず**最後に置きます。

図 20.1
CUA 標準メニューバー



次の図は，CUA が推奨しているメニュー項目群です。メニュー項目の中には，キーボードアクセラレータとして推奨されているキーがメニュー文字列の右に示されているものもあります。

図 20.2
CUA 推奨の File メニュー項目

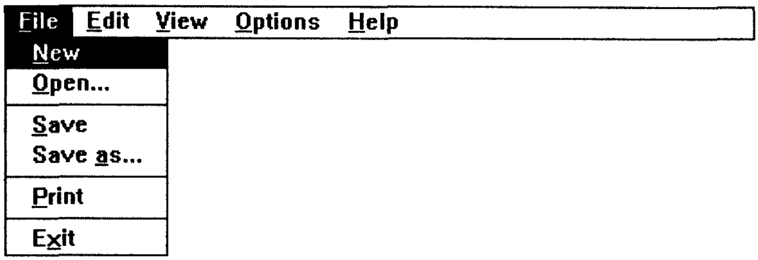


図 20.3
CUA 推奨の Edit メニュー項目

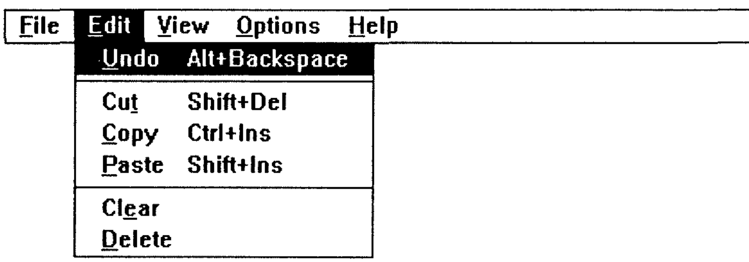


図 20.4
CUA 推奨の View メニュー項目

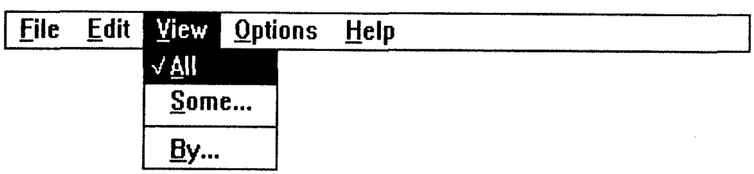


図 20.5
CUA 推奨の Options メニュー項目

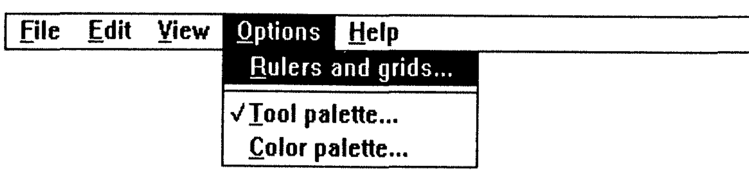
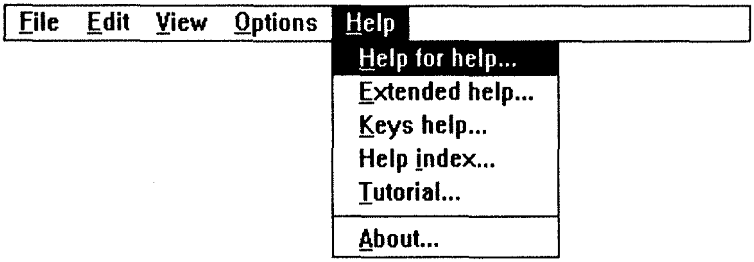


図 20.6
CUA 推奨の Help メニュー項目



ダイアログボックス

ダイアログボックスについては様々な設計方法があるため，汎用的な指針はほとんどありません。ただし，アプリケーションで使用するスタイルを決めたら，そのスタイルを一貫させるという考え方が大切です。ダイアログボックスには2つの種類があります。モードつきダイアログボックスとモードなしダイアログボックスです。モードつきダイアログボックスは最も多用され，ダイアログボックス表示中はプログラムの操作が一時中断されます。モードなしダイアログボックスでは，プログラムの通常動作は継続し，特殊な目的や文書のスペルチェックなど並行動作が必要な場合にのみ用いられます。

CUA ではファイルのオープン用ダイアログボックスと Save As ダイアログボックスに対し特定のスタイルを規定しています。ObjectWindows のストックダイアログはこの条件を満たしています。

設計時のその他の注意点

■ **適切なメインウィンドウサイズの指定**　メインウィンドウのサイズとして，必須情報の表示と使用するクライアント領域の表示に最低限必要なサイズを指定します。

- **ウィンドウの全面的再描画**　ウィンドウサイズを変更したあとでウィンドウを再描画する方法は2つあります。1つは，変更前の比率でウィンドウ内容を再描画する方法です。ウィンドウが小さすぎて全体が入りきらない場合はクリッピングされます。この方法は，WYSIWYG（what-you-see-is-what-you-get）アプリケーションに最も適しています。もう1つは，内容を再配置する方法です。常にウィンドウ内容がウィンドウ枠内にきちんと収められます。この方法は，ゲームやユーティリティに適しています。
- **表示色の一貫性**　表示色は，ユーザとアプリケーション間の情報のやりとりをさらに確かなものにします。表示色に一貫性を持たせるとともに，ユーザが表示色を変更できるようにします。

安全なプログラムの書き方

対話型のユーザインターフェースで生じるエラーの処理は，コマンドラインユーティリティの場合に比べて非常に複雑です。非対話型アプリケーションでは，エラー発生時にプログラムがエラーメッセージを表示して終了してもいっこうにかまいません（むしろ推奨できます）。しかし対話型の場合は，プログラムがエラーから回復し，ユーザからの入力を受け入れる体制に戻らなくてはなりません。エラーの性質がどのようなものであっても，エラーによってユーザが扱っている情報が破壊されたり，プログラムが突然終了するという事態は禁物です。このようなプログラミング上の基準を満たすプログラムは「安全」であるといえます。

ObjectWindowsを使えば安全なプログラムが書きやすくなります。複雑で察知しにくい**メモリ不足**エラーなどの検出やエラーからの回復を，容易にプログラミングすることができます。これは，アトム操作の概念を活用することによって可能になります。

アトム操作

アトム操作とは，これ以上細かい作業に分割できない単一の操作のことです。より厳密に言えば，結果として完璧な失敗と完璧な成功のどちらかしか得られない操作のことです。複雑な操作を基本的な機能単位に分解すると，メモリ管理が非常にやりやすくなります。

たいていのプログラムはメモリを小さな断片で多数確保します。たとえば，ダイアログボックスオブジェクトを構築する場合には，ダイアログボックス用のメモリと，ダイアログボックスが所有する他のオブジェクト(子コントロールなど)用のメモリを確保します。メモリ確保には毎回失敗の可能性があり，もし失敗した場合には，次のメモリ確保を行なうか，あるいは中止するかのチェックが必要になります。一連のメモリ割り当てのどこかで確保に失敗すると，それまで確保できた分のメモリも解放しなければならなくなります。とりあえず必要なすべてのメモリの確保を試み，そのあとでメモリ確保に失敗があったかどうかをチェックできれば理想的です。セーフティプールを利用すればこれが可能です。

セーフティプール

ObjectWindowsは，セーフティプールという固定長のメモリを事前に確保します。セーフティプールのデフォルトサイズは，SAFEPOOL.Hの***DEFAULT_SAFETY_POOL_SIZE***で定義します。ヒープ上に確保したメモリがセーフティプール領域に達すると，アプリケーションオブジェクトのメンバ関数***LowMemory***が真を返します。これは，これ以上メモリを確保しようとするのは危険であり，失敗する恐れがあることを示します。

セーフティプールのサイズは，個々のメモリ確保操作で確保しようとする最大のメモリサイズと同じでないと効果がありません。つまり，***LowMemory***関数が偽を返している限りは，メモリ確保は必ず成功するようになっていなければなりません。デフォルトのサイズはほとんどのアプリケーションで間に合うように設定してください。

アプリケーションオブジェクトは，メインウィンドウと子ウィンドウを生成した後で，メモリ不足かどうかのチェックをします。しかし，メインウィンドウの表示後に動的オブジェクトとしてウィンドウやダイアログを作成する場合は，そのつどメモリ不足チェック機構を明示的に呼び出す必要があります。メニュー選択の応答として，ダイアログボックスやポップアップウィンドウを開く場合も同様です。ただし，MDIウィンドウでは，子ウィンドウを生成した時点でメモリ不足チェックは終わっています。

ダイアログボックスオブジェクトのメンバ関数***Create***は，メモリ不足をチェックせずにウィンドウを作成するので，直接呼び出さないでください。



```
void TMyWindow::CMHelp(RTMessage Msg)
{
  PTHelpWindow HelpWindow;

  HelpWnd = new THelpWindow(this, "Help");
  HelpWnd -> Create();
}
```

上とは異なり，次のメンバ関数はメモリ不足をチェックします。



```
void TMyWindow::CMHelp(RTMessage Msg)
{
  PTHelpWindow HelpWindow;

  HelpWindow = new THelpWindow(this, "Help");
  if (GetApplication()->MakeWindow(HelpWindow))
    // ウインドウの作成成功
⋮
}
```

MakeWindow は，ウィンドウオブジェクトのポインタを返しますが，ウィンドウの作成に失敗した場合はヌルを返します。

モードつきダイアログの実行時にメモリ不足チェックを行なう場合は，次の例のように，***GetApplication*()->*ExecDialog*** を呼び出します。



```
void TMyWindow::CMRunDialog(RTMessage Msg)
{
  PTTestDialog TestDialog;
  int RetValue;

  TestDialog = new TTestDialog(this, "Test Dialog);
  RetValue = GetApplication()->ExecDialog(TestDialog);
  if ( Ret Value == IDCANCEL)
    // ダイアログの作成失敗、またはユーザによる取り消し
⋮
}
```

ダイアログの実行に成功すると，メンバ関数 ***ExecDialog*** はダイアログからの戻り値（通常は 0，または正の整数）を返します。ダイアログの実行に失敗するとメンバ関数 ***ExecDialog*** はエラー定数（負の整数）を返します。

MakeWindow と ***ExecDialog*** は，アプリケーションオブジェクトに対しメモリ不足状態を通知するため，定数 ***EM_OUTOFMEMORY*** を引数としてアプリケーションオブジェクトのメンバ関数 ***Error*** を呼び出します。***TModule*** で定義されているメンバ関数 ***Error*** は，ユーザにエラーを知らせるメッセージボックスを表示します。ObjectWindows では削除のたびにセーフティプールをチェックするようにグローバル演算子 delete を定義し直しています。セーフティプールが尽きると，delete 演算子は，セーフティプールの自動回復を行なってセーフティプールの再確保を試みます。セーフティプールの詳細については，**ObjectWindows for C++リファレンスガイド**の第 3 章『**ObjectWindows のその他の構成要素**』を参照してください。

ウィンドウ作成に関する他のエラー

通常のメモリ不足以外にも，ウィンドウやダイアログボックスの作成や表示を妨げる要因があります。たとえば，Windows メモリが不足したり，作成属性が無効であると，Windows ではインターフェース要素の生成を行なえません。

このようなメモリ不足以外の原因によるウィンドウ作成エラーは，インターフェースオブジェクトを生成する ObjectWindows のメンバ関数によって自動的に通知されます。***TWindowsObject*** には int データメンバ ***Status*** が定義されており，エラーが発生するとこの値が 0 以外になります。ObjectWindows で定義されているエラー定数の一部を以下に示します。

表 20.1
ウィンドウ作成エラーを示す値

エラー	意味
EM_OUTOFMEMORY	メモリ確保でセーフティプールに達した。
EM_INVALIDWINDOW	ウィンドウ要素を作成できなかった。
EM_INVALIDCLIENT	MDI ウィンドウオブジェクトがクライアントウィンドウを作成できなかった。
EM_INVALIDCHILD	オブジェクトの子ウィンドウでエラーが生じた。
EM_INVALIDMAINWINDOW	メインウィンドウを作成できなかった。

エラー定数とその値の一覧は，**ObjectWindows for C++リファレンスガイド**の第 3 章『**ObjectWindows のその他の構成要素**』の節「**EM_XXXX 定数**」にあります。

自作のウィンドウコンストラクタやウィンドウ作成メンバ関数で生じたエラーを検出できるようにするには，メンバ関数内で上に示した定数値の１つ，あるいは自分で定義した定数値を ***Status*** に代入します。たとえば，ウィンドウがファイルをオープンしようとして失敗した場合，ウィンドウは ***EM_FILEOPENERROR*** のような定数を用いてエラーを通知できます。***MakeWindow*** から呼び出された ***TWindow***::***Create*** が偽を返した場合，または ***ExecDialog*** から呼び出された ***TDialog***::***Execute*** が０以外を返した場合，呼び出した方のメンバ関数がアプリケーションのメンバ関数 ***Error*** にエラーを自動的に通知します。するとこのメンバ関数 ***Error*** は，デフォルトでアプリケーションクラスのメンバ関数 ***Error*** を呼び出します。ウィンドウクラスまたはアプリケーションクラスのメンバ関数 ***Error*** を定義し直して，各エラーに対応した処理を行なうこともできます。

```
TEditorWindow::TEditorWindow(PTWindowsObject AParent, LPSTR ATitle,
  LPSTR AFileName) : TWindow(AParent, ATitle)
{
  if ((Status == 0) && !OpenTheFile(AFileName))
    Status = EM_FILEOPENERROR;
}
```

大需要オブジェクト

大需要オブジェクトに関しては，メモリ不足を明示的にチェックする必要があります。大需要オブジェクトとは，セーフティプールのサイズよりも大きいメモリを確保するウィンドウオブジェクトのことです。また，ファイルの全内容をメモリに読み込ませる場合もメモリ不足のチェックが必要です。

大需要オブジェクトが自分自身の構築中にメモリを使い果たした場合は，データメンバ ***Status*** にエラー値を設定させ，それ以上メモリ確保を試みないようにさせてください。データメンバ ***Status*** はメンバ関数 ***MakeWindow*** で調べます。このメンバ関数はエラーを報告し，ウィンドウオブジェクトを廃棄します。

この方法では，コンストラクタが毎回支障なく働くようにするには不充分ですが，セーフティプールのサイズよりも大きなウィンドウの構築を管理する方法はこれ以外にはありません。

索　引

&文字　202

【A】

AddString
　TListBox メンバ関数　192
AddString 関数　348
AnimatePalette 関数　333
API
　Windows　110
Application 変数　315
AssignMenu
　TWindow メンバ関数　71, 237
Attr
　TWindow データメンバ　149
AutoMode
　TScroller データメンバ　161

【B】

BF_CHECKED 定数　214
BF_GRAYED 定数　214
BF_UNCHECKED 定数　214
BN_CLICKED 通知　213
Borland
　お問い合わせ　9
　カスタマーサポートセンター　9
BS_DEFPUSHBUTTON スタイル　212
BS_DIBPATTERN スタイル　312
BS_HATCHED スタイル　312
BS_HOLLOW スタイル　312
BS_PATTERN スタイル　312
BS_PUSHBUTTON スタイル　212
build
　ストリームクラスへのリンク　256

【C】

CanClose
　TApplication メンバ関数　46, 133
　TWindowsObject メンバ関数　46, 83, 133
　子ウィンドウと～　46
CanUndo
　TEdit メンバ関数　206
CBS_DROPDOWN スタイル　199
CBS_DROPDOWNLIST スタイル　199
CBS_SIMPLE スタイル　199
CBS_SORT スタイル　200
CF_TEXT 定数　297
Check
　TCheckBox メンバ関数　215
ChildMenuPos
　TMDIFrame データメンバ　242
Clear
　TEdit メンバ関数　206
　TStatic メンバ関数　203
ClearList
　TListBox メンバ関数　195
ClearModify
　TEdit メンバ関数　209
ClientWnd
　TMDIFrame データメンバ　241
CM_FIRST 定数　143, 343
CMEditCopy
　TEdit メンバ関数　207
CMEditUndo
　TEdit メンバ関数　207
CMFileNew
　TFileWindow メンバ関数　169
CMFileOpen
　TFileWindow メンバ関数　169
CMFileSave
　TFileWindow メンバ関数　169
CMFileSaveAs
　TFileWindow メンバ関数　169
CMTileChildren
　TMDIFrame メンバ関数　245
Common User Access（CUA）　353
Copy
　TEdit メンバ関数　206
Create
　MakeWindow 対～　78, 361, 362
　MakeWindow と～　152
　TDialog メンバ関数　60
　TWindowsObject メンバ関数　78, 137
　他メンバ関数に呼び出される～　152
CreateChild
　TMDIFrame メンバ関数　242

CreatePalette 関数　331
CreatePen 関数　60, 311
CreatePenIndirect 関数　311
CS_定数　155
CUA → Common User Access

【D】

DDE → ダイナミックデータエクスチェンジ
DDVT → ダイナミックディスパッチバーチャルテーブル
DefWindowProc 関数　120
DefWndProc
　TWindowsObject メンバ関数　120, 142
DeleteLine
　TEdit メンバ関数　210
DeleteObject 関数　349
DeleteSelection
　TEdit メンバ関数　210
DeleteString
　TListBox メンバ関数　194
DeleteSubText
　TEdit メンバ関数　210
DeltaPos
　TScrollBar メンバ関数　221
Destroy
　TWindowsObject メンバ関数　137
destructor
　TApplication　38, 132
DisableAutoCreate
　TWindowsObject メンバ関数　138
DisableTransfer
　TControl メンバ関数　228
DLL → ダイナミックリンクライブラリ
DOS　24

【E】

Editor
　TFileWindow データメンバ　168
EM_INVALIDCHILD 定数　363
EM_INVALIDCLIENT メンバ関数　363
EM_INVALIDMAINWINDOW メンバ関数　363
EM_INVALIDWINDOW 定数　363
EM_OUTOFMEMORY 定数　363
EnableAutoCreate
　TWindowsObject メンバ関数　139
EnableTransfer
　TControl メンバ関数　227
Error
　MakeWindow と～　362
ES_AUTOHSCROLL スタイル　206
ES_AUTOVSCROLL スタイル　206
ES_LEFT スタイル　206
ES_MULTILINE スタイル　206
ES_UPPERCASE スタイル　206
ExecDialog
　Error と～　363
　Execute 対～　82, 362
　TApplication メンバ関数　82, 174
　戻り値　362
Execute
　ExecDialog 対～　82, 362
_export
　コールバック関数と～　113

【F】

FirstThat
　TWindowsObject メンバ関数　139
ForEach
　TWindowsObject メンバ関数　139
fpstream　250

【G】

GDI　50 → グライフィックスデバイスインターフェース
　CreatePen 関数　60
　InvalidateRect 関数　53
　LineTo 関数　56
　MoveTo 関数　56
　描画ツール　57, 58
　　選択　58
　　デフォルト　58
　　ハンドル　58
　　ペイントと～　62

ペン
作成　60
GDI.EXE　24
GetCheck
TCheckBox メンバ関数　214
GetChildWithID
TWindow メンバ関数　188
GetCount
TListBox メンバ関数　194
GetLine
TEdit メンバ関数　207, 208
GetPaletteEntries 関数　332
GetPosition
TScrollBar メンバ関数　220
GetRange
TScrollBar メンバ関数　220
GetSelection
TEdit メンバ関数　209
GetSelIndex
TListBox メンバ関数　194
GetSelString
TListBox メンバ関数　194
GetStockObject 関数　310
GetString
TListBox メンバ関数　194
GetStringLen
TListBox メンバ関数　194
GetSubText
TEdit メンバ関数　209
GetText
TEdit メンバ関数　207
TStatic メンバ関数　203
GetWindowClass
TWindow メンバ関数　154
TWindowsObject メンバ関数　345
アイコンと～　345
オーバーライド　154
カーソルと～　345
GlobalAlloc 関数　276
戻り値　276
GlobalCompact 関数　280
GlobalDiscard 関数　279
GlobalFix 関数　278
GlobalFlags 関数　280
GlobalFree 関数　279
GlobalLock 関数　277
GlobalLRUNewest 関数　281
GlobalLRUOldest 関数　281
GlobalReAlloc 関数　279
GlobalSize 関数　277, 280
GlobalUnfix 関数　278
GlobalUnlock 関数　277
GlobalWire 関数　278
GMEM_DDESHARE 定数　297
GMEM_DDESHARE フラグ　280
GMEM_DISCARDABLE フラグ　276, 279
GMEM_FIXED フラグ　276, 278
GMEM_MODIFY フラグ　279
GMEM_MOVEABLE フラグ　276, 279
GMEM_NOCOMPACT フラグ　276, 279
GMEM_NODISCARD フラグ　277, 279
GMEM_NOT_BANKED フラグ　280
GMEM_ZEROINIT フラグ　279

【H】

HAccTable
TApplication データメンバ　344
hbrBackground
WNDCLASS メンバ　157
hCursor
WNDCLASS メンバ　156
hIcon
WNDCLASS メンバ　156
HideList
TComboBox メンバ関数　200
HWindow
TWindowsData メンバ　42

【I】

ID_FIRST 定数　96, 144
IDC_ARROW 定数　156
IDC_BEAM 定数　154
IDC_CROSS 定数　157
IDC_IBEAM 定数　157
IDC_WAIT 定数　157
IDE → 統合環境

IDI_定数　346
IDI_APPLICATION 定数　156
ID
　子ウィンドウ
　　範囲　144
　コントロール　94
　ダイアログボックスリソース　171
　メニュー項目　344
　メニュー　70
　文字列テーブル　347
　リソース　69
IDOK 定数　228
ifpstream　250
InitApplication
　TApplication メンバ関数　126, 130
InitChild
　TMDIFrame メンバ関数　243
InitClientWindow
　TMDIFrame メンバ関数　242
InitInstance
　TApplication メンバ関数　126, 129, 344
InitMainWindow
　TApplication メンバ関数　38, 126, 127
　マルチドキュメントインターフェースと
　　～　242
Insert
　TEdit メンバ関数　210
InsertString
　TListBox メンバ関数　193
INSTALL　4
InvalidateRect 関数　53
iopstream　250
ipstream　250
IsModified
　TEdit メンバ関数　209
IsVisibleRect
　TScroller メンバ関数　165

【K】

KERNEL.EXE　24

【L】

lbColor
　LOGBRUSH フィールド　313
LBN_SELCHANGE メッセージ　196
lbStyle
　LOGBRUSH フィールド　312
LineLength
　TEdit メンバ関数　207, 208
LineSize
　TScrollBar データメンバ　220
LineTo 関数　56
LMEM_DISCARDABLE フラグ　272, 274
LMEM_FIXED フラグ　272, 273, 274
LMEM_MODIFY フラグ　274
LMEM_MOVEABLE フラグ　272, 274
LMEM_NOCOMPACT フラグ　272, 274
LMEM_NODISCARD フラグ　272, 274
LMEM_ZEROINIT フラグ　274
LoadAccelerators 関数　344
LoadString 関数　347, 348
LocalAlloc 関数　272, 273
LocalCompact 関数　275
LocalDiscard 関数　274
LocalFlags 関数　275
LocalFree 関数　273
LocalLock 関数　273
LocalReAlloc 関数　274
LocalSize 関数　272, 275
LocalUnlock 関数　272
LOGBRUSH レコード　309
　定義　312
LOGFONT レコード　309
LOGPEN レコード　309
　定義　311
lopnColor
　LOGPEN フィールド　310, 311
lopnStyle
　LOGPEN フィールド　311
lopnWidth
　LOGPEN フィールド　311
LowMemory 関数　361
LParam
　メッセージ引数　142
lpszMenuName
　WNDCLASS メンバ　157

LRU → メモリ管理（LRU）

【M】

MainWindow
　TApplication データメンバ　38
MakeWindow
　Create 対～　78, 361
　Create と～　152
　Error と～　363
　TApplication メンバ関数　78, 362
　TWindowsObject メンバ関数　152
　戻り値　78
MDI → マルチドキュメントインターフェース
Menu
　TWindowAttr フィールド　71
MessageLoop
　TApplication メンバ関数　126, 132
　簡単な～　132
MoveTo 関数　56

【N】

NF_FIRST 定数　222
NumLines
　TEdit メンバ関数　207, 208

【O】

ObjectWindows　16
　DOS のヘルプ　6
　IDE と～　34
　Windows のヘルプ　5
　インストール　4
　オンラインヘルプ　6
　階層　102
　規約　101
　コマンドラインツールと～　34, 36
　プロジェクトと～　34
　ヘッダファイル　31
　メイクファイルと～　34
OBM_CLOSE ビットマップ　349
OBM_DNARROW ビットマップ　349
OBM_ZOOM ビットマップ　349
ofpstream　250
opstream　250

【P】

PageSize
　TScrollBar データメンバ　220
Paint
　TWindow メンバ関数　62, 66
　ウィンドウのスクロールと～　160, 164
Parent
　TWindowsData メンバ　43, 138
PC_RESERVED 定数　333
PostMessage 関数　289
ProcessAccels
　TApplication メンバ関数　132
ProcessDlgMsg
　TApplication メンバ関数　132
ProsessMDIAccels
　TApplication メンバ関数　132
pstream
　ストリームオブジェクトと～　248

【R】

.RC ファイル　338
.RES ファイル　70
read
　ストリームクラスへのリンク　256
README ファイル　4
RealizePalette 関数　331, 333
RegisterWindowMessage 関数　289
ReleaseCapture 関数　57
Result
　TMessage メンバ　142
　メッセージ引数　142
RGB マクロ　312
.RES ファイル　336
RTMessage 型　52, 141
Run
　TApplication メンバ関数　126, 132

【S】

SBBottom
　TScrollBar メンバ関数　223
SBLineDown

TScrollBar メンバ関数 223
SBLineUp
TScrollBar メンバ関数 223
SBPageDown
TScrollBar メンバ関数 223
SBPageUp
TScrollBar メンバ関数 223
SBS_HORZ スタイル 219
SBS_TOPALIGN スタイル 219
SBS_VERT スタイル 219
SBThumbPosition
TScrollBar メンバ関数 223
SBThumbTrack
TScrollBar メンバ関数 223
SBTop
TScrollBar メンバ関数 223
Scroll
TEdit メンバ関数 211
ScrollBy
TScroller メンバ関数 163
ScrollTo
TScroller メンバ関数 163
SelectPalette 関数 331
SendDlgItemMessage 関数 121
SendDlgItemMsg
TDialog メンバ関数 121, 180
SendMessage 関数 121, 289
SetCapture 関数 57
SetCheck
TCheckBox メンバ関数 214, 215
SetPageSize
TScroller メンバ関数 163
SetPaletteEntries 関数 333
SetPosition
TScrollerBar メンバ関数 221
SetRange
TScrollBar メンバ関数 220
TScroller メンバ関数 162
SetSelection
TEdit メンバ関数 211
SetSelIndex
TListBox メンバ関数 195
SetSelString
TListBox メンバ関数 195
SetText
TEdit メンバ関数 211
TStatic メンバ関数 203
SetupWindow
TMDIFrame メンバ関数 242
TWindow メンバ関数 95
TWindowsObject メンバ関数 138
Show
TWindowsObject メンバ関数 137
ShowList
TComboBox メンバ関数 200
SS_NOPREFIX スタイル 202
SS_SIMPLE スタイル 203
Status
TWindowsObject メンバ関数 363
エラーの通知と～ 364
streamableName
ストリームクラスと～ 255
Style
TWindowAttr データメンバ 151

【T】

TApplication
hInstance データメンバ 26
hPrevInstance データメンバ 26
lpCmdLine データメンバ 26
nCmdShow データメンバ 27
Status データメンバ 38
データメンバ 26
必須条件 37
TApplication クラス 104, 125
TButton オブジェクト 212
TCheckBox オブジェクト 213
TComboBox オブジェクト 194
TControl クラス 92
TEdit オブジェクト 204
TEditWindow クラス 165
TextOut 関数 52
TF_GETDATA 定数 228
TF_SETDATA 定数 228
TFileDialog クラス 183
TFileDlgRec レコード 79

TFileWindow クラス　165
THELP　6
　読み込み　6
TileChildren
　TMDIFrame メンバ関数　245
TInputDialog コンストラクタ　181
TListBox オブジェクト　191
TMDIClient オブジェクト　241
TMDIFrame オブジェクト　241
TMessage 構造体　141
TMessage 型　52
TModule クラス　104, 125
Toggle
　TCheckBox メンバ関数　215
TRadioButton オブジェクト　213
Transfer
　TControl メンバ関数　229
　TWindowsObject メンバ関数　229
　戻り値　229
TransferBuffer
　TDialog データメンバ　227
　TWindow データメンバ　227
TransferData
　TWindowsObject メンバ関数　228
TScrollBar オブジェクト　219
TScrollBarData 構造体　226
TScroller
　ウィンドウでの使用　159
　行サイズ　159
　スクロール単位　159, 162
　属性　158
　ページサイズ　158
TScroller クラス　158
TStreamable　85
　ストリームマネージャと～　252
TStreamableClass
　ストリームマネージャと～　252
TStreamableTypes
　ストリームマネージャと～　252
TWindow　40, 41
TWindow クラス　148
TWindowAttr 構造体　149
TWindowsObject　41
TWindowsObject クラス　135

【U】

Uncheck
　TCheckBox メンバ関数　215
Undo
　TEdit メンバ関数　206
UpdateColors 関数　334
USER.EXE　24

【W】

Windows
　API　110
　　ObjectWindows と～　112
　　ハンドルと～　110
　　呼び出し　18
　構造　24
　座標系　52
Windows メッセージ　116
WM_CLOSE メッセージ　133
WM_COMMAND メッセージ　142
WM_COMPACTING メッセージ　281
WM_DDE_ACK メッセージ　293, 297
WM_DDE_ADVISE メッセージ　295
WM_DDE_DATA メッセージ　295, 297
WM_DDE_EXECUTE メッセージ　295
WM_DDE_INITIATE メッセージ　292
WM_DDE_POKE メッセージ　295
WM_DDE_REQUEST メッセージ　295, 297
WM_DDE_TERMINATE メッセージ　292
WM_DDE_UNADVISE メッセージ　295
WM_HSCROLL メッセージ　221
WM_LBUTTONDOWN メッセージ　44
　引数　52
WM_LBUTTONUP メッセージ　55
WM_MDIACTIVATE メッセージ　245
WM_MOUSEMOVE メッセージ　55
WM_PALETTECHANGED メッセージ　334
WM_RBUTTONDOWN メッセージ　44
WM_SETFOCUS メッセージ　166
WM_VSCROLL メッセージ　221

WMSetFocus
　TEditWindow メンバ関数　166
WMSize
　TEditWindow メンバ関数　166
WNDCLASS
　デフォルト値　155
WNDCLASS 構造体　155
WParam
　メッセージ引数　142
write
　ストリームクラスへのリンク　256
WS_CHILD スタイル　200
WS_HSCROLL 定数　159
WS_HSCROLL スタイル　206
WS_VISIBLE スタイル　137, 200
WS_VSCROLL 定数　159
WS_VSCROLL スタイル　200, 206

【X】

XUnit
　TScroller データメンバ　159

【Y】

YUnit
　TScroller データメンバ　159

【あ】

アイコン
　ウィンドウ　153
　カーソル対～　345
　ストック　346
　デフォルト　155
アクセラレータ
　MDI
　　メッセージと～　132
　コマンドと～　344
　定義　69
　メッセージと～　132
　ロード　129, 344
圧縮（メモリ）　268
　例　268
アトム
　ダイナミックデータエクスチェンジと～
　　290
アプリケーション　13
　"Hello, World"　25
　Status　38
　安全　360
　オブジェクト　37, 104, 125, 353
　開発　28
　起動時の処理　26
　クローズ　132
　グローバル変数　363
　構造　24
　構築　30
　コンストラクタ　38, 126
　コントロールメニュー　128
　作成の指針　353
　終了　46
　初期化　126
　　各インスタンス　126, 129
　　最初のインスタンス　126, 130
　設計　353
　第1インスタンス　129
　デストラクタ　38
　名前　38
　必須条件　16, 37
　メインプログラム　125
　有益性　15
アンド（&）文字　202
安全なプログラミング　78, 82, 360
　Status メンバと～　364
　ウィンドウ作成と～　363
　ダイアログボックスと～　362
　大需要オブジェクト　364
　メモリと～　361

【い】

イベント　18
　コントロール
　　応答　96
　メッセージと～　140
　メニュー　69
イベント駆動型のプログラミング　140

色
RGB 312
輝度 311
指定 312
使用 360
ディザリング 307
ディスプレイコンテキストと～ 306, 308
バックグラウンド～ 153, 306, 309
ウィンドウ 157
ブラシ 312
ペン 311
インスタンスサンク → コールバック関数
インスタンスリンケージ 39
インターフェース
オブジェクト 42, 105, 136
Status 363
ウィンドウ 148
コンストラクタ 136
コントロール 178, 186
廃棄 136
必要な理由 136
要素 42
作成 78, 136
失敗 363
対応づけ 78
破棄 137
表示 137
インターフェースオブジェクト 17
インターフェース要素 17

【う】

ウィンドウ
エディット → エディットウィンドウ
オブジェクト 41, 105
派生 43
親 42 → 親ウィンドウ
クラス → クラス（ウィンドウ）
クローズ 97
子～ 42 → 子ウィンドウ
コンストラクタ 71
コントロール → コントロール
サイズ変更 359
作成
失敗 363
作成する
セーフティプールと～ 361
作成属性 153
座標 52
初期化（コントロール） 227
スクロール 157 → スクロール（ウィンドウ）
スタイル 149, 151
デフォルト 150
属性
作成 153
デフォルト 150
登録 153, 345
メニュー 71
ダイアログ → ダイアログウィンドウ
ダイアログボックス対～ 175
タイトル 149
転送機構 227
～内のコントロール 91
バックグラウンド色 157
表示 137
標準 165
ファイル → ファイルウィンドウ
描画 360
ポップアップ 75, 87
ダイアログボックス対～ 78
追加 76
メイン → メインウィンドウ
ウィンドウオブジェクト 147
ウィンドウ要素と～ 148
初期化 148
属性 148
ウィンドウ要素
作成 152

【え】

エディット（Edit）ウィンドウ 165, 166
File ウィンドウと～ 168
エディットコントロールと～ 166
エディットコントロール 203
Edit ウィンドウと～ 166

クリップボードと～　206
構築　204
照会　207
消去　210
スクロール　211
スタイル　205
変更　210
テキスト
行の取得　210
削除　210
取得　211
設定　211
選択　211
選択～
削除　210
取得　210
ステータス　210
挿入　210
フォーマット　209
複数行
行の長さ　208
転送バッファ　227
入力フォーカスを持つ～　211
複数行～　204, 208
行の削除　210
サイズ　208
スクロール　211
変更　210
メニューコマンドと～　206, 207, 211
例　211
エラー
処理　363
演算子
連結できる～　249
演算子＜＜
ストリームごとのオーバーロード　249
ヘッダファイル　251
演算子＞＞
ストリームごとのオーバーロード　249

【お】

オブジェクト → 描画ツール
MDI　109
アプリケーション　37, 104, 125
インターフェース　17, 42, 105
ウィンドウ　41, 105, 147
派生　43
コントロール　107
所有　37
スクローラ
ダイアログ　106
保管　247
モジュール　104, 125
親ウィンドウ　42, 75, 137
子リスト　138
上位型対～　76
割り当て　150

【か】

カーソル
I 字型　156, 157
アイコン対～　345
十字型　157
ストック　156, 346
砂時計型　157
デフォルト　157
変更　156
マウス　157
下位項目 → メニュー（下位項目）
カスタマーサポートセンター（Borland）　9
カスタムコントロール　231
画面
クリア　53, 74
カラーパレット 330 → パレット
関数
API
～の種類　112
コールバック～　113

【く】

クライアントウィンドウ → マルチドキュメントインターフェース

クライアント領域
　スクロール　158, 160
　定義　51
クラス
　ウィンドウ～　153
　　～の登録　153
　　～の命名　153
　スタイル　156
　属性　345
　登録　153
　リソースと～　345
グラフィックス　305
グラフィックスデバイスインターフェース
　305
繰り返し処理メンバ関数
　子ウィンドウ　139
クリッピング　308
クリップボード　283
　エディットコントロールと～　206
　データ
　　～の格納　284
　　～を取り込む　286
　データ形式　283, 296
　　CF_TEXT　297
　　レンダリング　288
グループボックス　216
　構築　216
　選択ボックスと～　214
　メッセージと～　216
　例　217
グローバルヒープ　270

【こ】

コールバック関数　113
　_export と～　113
子ウィンドウ　42, 75, 138
　ID の範囲　144
　構築　138
　作成　138
　　～の無効化　138
　従属～　75
　　～のコントロール　91
　所有と～　76
　属性
　　～の設定　151
　データメンバ　93
　独立　75
　破棄　137
　マルチドキュメントインターフェース
　239
構成
　ソフトウェア　3
　ハードウェア　3
コマンド
　アクセラレータと～　344
　メニュー
　　エディットコントロールと～　207
コマンドボタン → プッシュボタン
子リスト　138
　TMDIFrame データメンバ　241
　TWindow データメンバ　43, 138
コンストラクタ
　build　254
　TApplication　38, 126
　TButton　213
　TCheckBox　213
　TComboBox　199
　TDialog　175
　TEdit　204
　TFileDialog　183
　TFileWindow　165, 168
　TGroupBox メンバ関数　216
　TInputDialog　181
　TListBox　191
　TRadioButton　213
　TScrollBar　219
　TScroller　159
　TStatic　202
　TWindow　71
　TWindowsObject　136
コンテナクラスライブラリ
　Array　63
　ArrayIterator　65
　Object　63
　Object クラス　104
コントロール　91, 185

ID　94
値
　設定　224
イベント　96
ウィンドウ　91
ウィンドウと～　186
ウィンドウへの挿入　87
エディット → エディットコントロール
オブジェクト　107, 186
　親　94
　コンストラクタ　94
　操作　177
　ダイアログボックスと～　177
　表示　95
オブジェクトとの対応づけ　178
管理　94
グループボックス → グループボックス
子ウィンドウ　177
コンボボックス → コンボボックス
従属子ウィンドウ　91
初期化　227
スクロールバー → スクロールバー
スタティック → スタティックコントロール
ダイアログボックスと～　180
通知メッセージ　179, 190
　引数　190
チェックボックス → チェックボックス
破壊　188
プッシュボタン → プッシュボタン
ボタン → プッシュボタン
メッセージ　96
メッセージと～　121, 179, 188
ラジオボタン → ラジオボタン
リストボックス → リストボックス
～へメッセージ送出　179
リソースと～　92, 186
コントロールオブジェクト　185
　操作　188
コントロールメニュー
　アプリケーション　128
コンボボックス　197
　アプリケーション例　200
　構築　199
　種類　197
　シンプル　198
　スタイル　199
　転送バッファ　227
　ドロップダウン　198
　ドロップダウンリスト　198
　変更　200
　リスト
　　表示　200
　　非表示　200
　リストボックス対～　197

【さ】

座標
　物理　308
　変換　308
　論理　308
座標系
　Windows　51
　ディスプレイコンテキスト　308
　デフォルト　308
サンク → インスタンスサンク

【し】

自動スクロール　161
寿命の長いオブジェクト　247
情報の取得　332
ショートカット　333 → アクセラレータ

【す】

スクロール（ウィンドウ）　157, 159
　Paint と～　160, 164
　位置
　　変更　163
　構築　159
　自動スクロール　161
　スクロールバーと～　160
　単位
　　変更　162
　トラッキング　161
　範囲
　　変更　162

ページサイズ
変更 164
例 159
スクロールバー 218
位置 220
設定 221
ウィンドウのスクロールと～ 159
構築 219
自動スクロールと～ 161
照会 220
サイズ
デフォルト 219
スタイル 218
通知メッセージと～ 221
転送バッファ 227
範囲 220
設定 221
ページサイズ 220
変更 220
ラインサイズ 220
例 223
スクローラ
オブジェクト 109
図形
塗りつぶし 309
描画 309
スタイル 112
WNDCLASS メンバ 156
ウィンドウ 149, 151, 191
エディットコントロール 205
組み合わせ 112
クラス 156
コンボボックス 197
スクロールバー 219
スタティックコントロール 195
プッシュボタン 212
ブラシ 312
ライン 311
リストボックス 192
論理ペン 311
スタティックコントロール
構築 202
照会 203
消去 203
スタイル 201
デフォルト 203
転送バッファ 225
変更 203
文字
下線つき 202
ストリーム 247
定義 248
～からの読み込み 254
～への書き込み 254
ストリームオブジェクト 85
read メンバ関数 86
write メンバ関数 86
構築 254
ストリームクラス
build メンバ関数 254
pstream と～ 248
read メンバ関数 254
TWindowsObject と～ 248
write メンバ関数 254
コンストラクタ 254
作成 247, 257
使用 258
定義 251
登録 256, 257
命名 255
メモリオーバーヘッド 252
ライタ
定義 253
リーダ
定義 252
ストリームマネージャ
TStreamable と～ 252
TStreamableClass と～ 252
TStreamableTypes と～ 252
コード
リンク 256
スマートコールバック
FreeProcInstance 対～ 115
MakeProcInstance 対～ 115
列挙関数と～ 115

【せ】

セーフティプール　78, 82, 361
　　サイズ　361
選択ボックス → チェックボックス，ラジオボタン
　　グループ　213
　　構築　213
　　照会　214
　　状態
　　　　変更　214
線
　　描画　309

【そ】

属性
　　作成　153
　　登録　155, 345
　　　　アイコン　156
　　　　カーソル　156
　　　　スタイル　156
　　　　デフォルトメニュー　157
　　　　バックグラウンド色　157

【た】

ダイアログ
　　オブジェクト　106
　　入力 → 入力ダイアログ
　　ファイル → ファイルダイアログ
ダイアログボックス　75, 78, 171
　　値を返す　79
　　ウィンドウ対～　175
　　オブジェクト
　　　　構築　172
　　クローズ　174
　　コントロールと～　180
　　コントロールの初期化　227
　　コントロールの読み取り　227
　　作成　173
　　実行　81, 173
　　　　安全なプログラミングと～　362
　　使用　172
　　ストック　79
　　転送機構　227
　　汎用　359
　　ファイル　79
　　　　実行　81
　　ポップアップウィンドウ対～　79
　　モードつき　174
　　モードなし
　　　　メッセージと～　132
　　文字列テーブルと～　347
　　読み込み　345
　　リソースと～　78, 171, 345
タイトル
　　メインウィンドウ　130
ダイナミックデータエクスチェンジ　290
　　アトムと～　290
　　対話
　　　　終了　294
　　　　成立　291
　　データ
　　　　継続転送の維持　299
　　　　単一項目の要求　295
　　トピック
　　　　システム　303
　　ホットリンク　295, 299
　　マクロ
　　　　実行　301
　　メッセージ　291
　　用語の定義　290
ダイナミックディスパッチバーチャルテーブル（DDVT）　17
ダイナミックディスパッチメンバ関数
　　制約　20
単位
　　スクロール　158, 162
　　ディスプレイ　308

【ち】

チェックボックス 213 → 選択ボックス
　　切り換え　215
　　スタイル
　　　　デフォルト～　213
　　状態
　　　　～の設定　215

チェックする　215
チェックなしにする　215
転送バッファ　227
3 つの状態
～の切り換え　215
例　217

【つ】

通知コード　179

【て】

データメンバ
オブジェクト
追加　80
子ウィンドウと～　93
定数
スタイル　112
～の組み合わせ　112
ディスパッチインデックス　19
ディスプレイコンテキスト　49, 306
解放　50
関数　50
管理　306
座標　308
取得　49
単位　308
使う　49
デバイスドライバと～　305
バックグラウンド色　309
ハンドル　49
ビットマップと～　307
描画ツール　309
ペイントと～　62
テキスト
描画　50, 309
編集　165
デフォルト
プッシュボタン　212
デフォルトのメッセージ処理　120, 142, 144
転送バッファ　225
エディットコントロール　227
更新　228
コンボボックス　226
使用　228
スクロールバー　226
スタティックコントロール　225
チェックボックス　226
定義　225
ラジオボタン　226
リストボックス　226
転送機構　224
ウィンドウ　227
カスタムコントロールと～　229
自動　228
ダイアログボックス　227
手作業　228
例　229

【と】

登録
ウィンドウクラス　153, 154
ストリームクラス　256
属性
ウィンドウ　154
トラッキング
ウィンドウのスクロールと～　161
ドラッグ　54

【な】

名前
アプリケーション　38

【に】

入力ダイアログ　60, 181

【は】

バックグラウンド色
ウィンドウ　157
ディスプレイコンテキスト　309
パレット　307, 330
変更　333, 334
～を用いた描画　332
範囲
ウィンドウのスクロール　162
スクロールバー　218
ハンドル

ウィンドウ　42, 137
ディスプレイコンテキスト　49
描画ツール　58
メモリ　269

【ひ】

ビルダ　254
ビルドコンストラクタ　254
描画
テキスト　51
描画ツール　57, 58, 307, 309 - 317
ストック　309, 310
一覧　310
選択　58
属性　309
デフォルト　58
ハンドル　58
ペイントと～　62
論理　309, 310
表記（マニュアルの～）　7
ビットマップ
絵　349
機種非依存型　307
互換性　307
削除　349, 351
ストック　349
ディスプレイコンテキストと～　307
ブラシ　349, 350
サイズ　350
メニュー項目　349, 351
読み込み　348

【ふ】

ファイル
. RC　338
. RES　336
オープン　79, 184
テキスト
編集　165, 168
ヘッダ　341
保管　79, 184
リソース　69
リソースコンパイラのスクリプト　337
ファイル（File）ウィンドウ　165, 168
Edit ウィンドウと～　168
～の使用　168
ファイルダイアログ　183
構築　80
実行　81
マスク → ファイル（マスク）
フォント
ディスプレイコンテキストと～　306, 309
プッシュボタン
構築　212
スタイル　212
デフォルト　212
メッセージと～　212
例　217
フラグ
グローバルメモリ　276
ローカルメモリ　272
ブラシ
色　312
削除　351
種類　309
ディスプレイコンテキストと～　309
何も描かない～　312
ハッチ　312
ビットマップ　313, 349, 350
論理～　312
作成　312
スタイル　312
フレームウィンドウ → マルチドキュメントインターフェース
プロセス間通信　283

【へ】

ペイント　62, 66
ディスプレイコンテキストと～　62
描画ツールと～　62
ヘッダファイル　341
ObjectWindows　31
ペン
作成　60
線と～　309

ディスプレイコンテキストと～　306
ラインスタイル　311
論理　311
色　311
作成　311
スタイル　311
幅　311

【ほ】

ホットキー → アクセラレータ
ホットリンク　295

【ま】

マウスクリック
座標　52
マッピングモード　308
マルチタスク
メモリ管理と～　267
マルチドキュメントインターフェース　239
ウィンドウ
構築　241
オブジェクト　109,241
クライアントウィンドウ　239,241
構築　242
子ウィンドウ　239,240
Tile　245
アクティブにする　244
作成　242,244
管理　244
タイトル　240
コマンドメッセージと～　245
フレームウィンドウ　239,240
メインウィンドウ　242
メニュー　242
メッセージと～　244
メニューと～　245
要素　239
例　245

【め】

命名規約　101
メインウィンドウ　27,36,37
MDIアプリケーション　242
アプリケーションオブジェクトと～
125
カスタマイズ　43
クローズ　129
最小化　128
サイズ　150,359
最大化　128
作成　38
初期化　126,127
タイトル　129,130
特性　127
メニューと～　71
役割　25
メッセージ　18,27,116
Windows　116
イベントと～　116,140
ウィンドウ　121
応答する　44,141
～メンバ関数　44
応答メンバ関数　141
グループボックスと～　216
結果の値　142
子ID～　122,142,144,179,212
マルチドキュメントインターフェース　244
構造体　52
コマンド　72,122
コマンド～　142
マルチドキュメントインターフェース　245
コントロール　96,179
デフォルト処理　98
コントロール通知　179
引数　179
コントロール通知～　190
コントロールと～　121,188
スクロールバーと～　221
種類　118
処理　126,132,140
デフォルト　142
送出　121
通知　121,144
デフォルト処理　98,120,144

範囲　122
引数　52,117,142
プッシュボタンと～　212
プロセス間　289
マウス
入力の取得　57
マウスイベント　44,55
マウスのドラッグ　55
マルチドキュメントインターフェースと～　244
メニューと～　72
ユーザ定義～　122
リストボックス通知～　195
ルーティング　141
メッセージ応答関数　19
メッセージ応答メンバ関数　19
メニュー　69
ID　70
イベント　69
オブジェクトと～　71
下位項目　69
項目
ID　344
ショートカット　344
コマンド
応答　343
最上位　69
デフォルト　157
ビットマップと～　349,351
標準　357
メインウィンドウと～　71
メッセージと～　72
文字列テーブルと～　347
読み込み　71,343
リソース　69
メニューコマンド
エディットコントロールと～　207
メモリ　268
圧縮～　268
圧縮する　281
安全な確保　361
大きなメモリ　271
共有　282
グローバル　270,276
64 KB 境界　279
圧縮　280
大きさ　280
解放　279
共有　282
固定　278
ロック　278
情報の取得　280
～のサイズ　276
廃棄　279,281
ハンドル
無効　277
フラグ　276,280
～へのポインタ　277
ロック　277
ロック解除　277
ストリームクラスと～　251
セーフティプール　361
大需要オブジェクト　364
廃棄可能　268
ローカル　270,272
圧縮　275
移動可能　272
解放　273
ロック回数と～274
確保　272
固定　272,273
再確保　274
最大サイズ　272
設定　272
廃棄　273
ロック回数と～　274
廃棄可能　272
ハンドル　273
無効～　273
フラグ　272,275
～の取得　275
ロック　272
ロック解除　272
～を指すポインタ　272
ローカル対グローバル　270
ロック　269

ロック解除　269
ロック回数　272
メモリ管理　267
LRU アルゴリズム　281
ハンドルと～　269
プロテクトモードと～　270
マルチタスクと～　267
メモリ不足に関する警告　281
リアルモードと～　270
メンバ関数　19
コマンド応答～　142
メッセージ応答～　141
メッセージの応答　44

【も】

モード
マッピング → マッピングモード
モードなしダイアログ
ウィンドウ対～　174
使用　174
メインウィンドウ対～　175
モジュール
ObjectWindows と～　88
オブジェクト　104, 125
文字列テーブル
ID　347
ダイアログボックスと～　347
メニューと～　348
読み込み　347

【ゆ】

有益性
アプリケーション　15
開発者にとっての～　15
ユーザにとっての～　15
ユーザインターフェースオブジェクト → インターフェースオブジェクト

【よ】

要素
インターフェース　17, 42

【ら】

ライタ　254
定義　252
ライブラリ → ダイナミックリンクライブラリ
Borland C++　31
ObjectWindows　30
インポート　32
購入した～　31
コンテナクラス　30
メモリモデルと～　32
ラインスタイル　311
ラジオボタン　213 → 選択ボックス
スタイル
デフォルト　213
チェックつきにする　215
チェックなしにする　215
転送バッファ　226
例　217

【り】

リーダ　254
定義　252
リストボックス　191
イベントと～　192
構築　191
照会　194
選択　195
ソートしない～　192
ソートする　192
通知メッセージ　192
デフォルトスタイル　192
転送バッファ　226
変更　192
文字列
項目数　192
削除　192
取得　192
全体の消去　192
挿入　193
追加　192
長さ　192

例　195
リソース　335
ID　70
アイコン　345
カーソル　154, 345
コントロールと～　186
削除　351
作成　69, 336
実行形式ファイルへの追加　336
種類　335
ダイアログボックス　171, 345
ID　172
名前　343
廃棄　335
ビットマップ　349
ファイル　70, 336
ヘッダファイルと～　341
メニュー　69, 343
デフォルト　157
文字列　347
読み込み　335, 336, 342, 343
リソースコンパイラ　337
コマンドラインオプション　337
使用上の注意　341
スクリプトファイル　337, 338
リソースファイル
作成　33
領域
クリッピング　308

【れ】

列挙関数　114

【ろ】

ローカルヒープ　271
最大サイズ　271

ObjectWindows^{TM} for C++　プログラマーズガイド

1992年　1月　6日　初版発行

著者　**Borland International**

編訳・発行　**ボーランド株式会社**

〒107 東京都港区南青山 7-8-1　小田急南青山ビル
TEL　(03) 3486-1400(代表)
FAX　(03) 3486-1423

Printed in Japan

落丁，乱丁はお取り替えいたします。


BORLAND® C++
BORLAND
Copyright©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND
Copyright©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND
Copyright©1987, 1991 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. Unauthorized use, duplication or distribution is strictly prohibited by federal law.
APPLICATION FRAMEWORKS
BORLAND® C++
BORLAND

PROGRAMMER'S GUIDE

OBJECTWINDOWS™ FOR C++

BORLAND